Federating IoT and cloud infrastructures to provide scalable and interoperable Smart Cities applications, by introducing novel IoT virtualization technologies

EU Funding: H2020 Research and Innovation Action GA 814918; JP Funding: Ministry of Internal Affairs and Communications (MIC)

Deliverable 4.1
Smart-city information sharing services - First release

Deliverable Type:	Report
Deliverable Number:	4.1
Contractual Date of Delivery to the EU:	31.12.2019
Actual Date of Delivery to the EU:	31.12.2019
Title of Deliverable:	Smart-city information sharing services - First release
Work package contributing to the Deliverable:	WP4
Dissemination Level:	Public
Editor:	Frank Le Gall (EGM), Hidenori Nakazato (WAS)
Author(s):	Giuseppe Tropea, Andrea Detti, Ludovico Funari (CNIT); Ahmed Abid, Benoit Orihuela, Hamza Baqa (EGM); Hidenori Nakazato, Kenji Kanai (WAS); Juan A. Martinez, Juan A. Sanchez, Antonio Skarmeta (OdinS), Kenichi Nakamura (PAN), Martin Bauer (NEC)
Internal Reviewer(s):	Andrea Detti (CNIT)
Abstract:	This deliverable describes the first release of the information sharing services of Fed4IoT, including the architecture set-up, the information model used for federation, the API provided and the security mechanism implemented.
Keyword List:	Information sharing; Federation; Security

Disclaimer

This document has been produced in the context of the EU-JP Fed4IoT project which is jointly funded by the European Commission (grant agreement n° 814918) and Ministry of Internal Affairs and Communications (MIC) from Japan. The document reflects only the author's view, European Commission and MIC are not responsible for any use that may be made of the information it contains

Table of Contents

List of Figures

List of Tables

Abbreviations

Abbreviation	Definition
ADN	Application Dedicated Node
AE	Application Entity
AIMD	Additive Increase/Multiplicative Decrease
API	Application Programming Interface
ASM	Adaptive Semantic Module
ASN	Application Service Node
AWS	Amazon Web Services
CIM	Context Information Management
CSE	Common Services Entity
ETSI	European Telecommunications Standards Institute
FIB	Forwarding Information Base
GE	Generic Enabler
HTTP	HyperText Transfer Protocol
ICN	Information Centric Networks
ICT	Information and Communication Technologies
IN	Infrastructure Node
IP	Internet Protocol
ISG	Industry Specification Group
JSON	JavaScript Object Notation
MANO	MAnagement and Network Orchestration
MMG	Morphing Mediation Gateway
MN	Middle Node
MQTT	Message Queue Telemetry Transport
NGSI	Next Generation Service Interfaces Architecture
NGSI-LD	Next Generation Service Interfaces Architecture - Linked Data
NSE	Network Service Entity
OMA	Open Mobile Alliance
PIT	Pending Interest Table
PPP	Public-Private Partnership
RDF	Resource Description Framework
REST	Representational State Transfer
SDK	Software Development Kit
TCP	Transmission Control Protocol
TM	Topology Master
TN	Task Name
TV	ThingVisor
UML	Unified Modeling Language
URI	Uniform Resource Identifier

VNF	Virtual Network Functions
vSilo	Virtual Silo
vThing	Virtual thing
WLAN	Wireless Local Area Network

Table 1: Abbreviations

Fed4IoT Glossary

Table 2 lists and describes the terms that have been considered relevant in this deliverable.

Term	Definition
FogFlow	An IoT edge computing framework that automatically orchestrates dynamic data processing flows over cloud- and edge-based infrastructures. Used for ThingVisor development
Information Centric Networking	New networking technology based on named contents rather than IP addresses. Used for ThingVisor development
IoT Broker	Software entity responsible for the distribution of IoT information. For instance, Mobius and Orion can be considered as Brokers of oneM2M and FIWARE IoT platforms, respectively
Neutral Format	IoT data representation format that can be easily translated to/from the different formats used by IoT brokers
Real IoT System	IoT system formed by real things whose data is exposed trough a Broker.
System DataBase	Database for storing system information
ThingVisor	System entity that implements Virtual Things
VirIoT	Fed4IoT platform providing Virtual IoT systems, named Virtual Silos
Virtual Silo (vSilo)	Isolated virtual IoT system formed by Virtual Things and a Broker
Virtual Silo Controller	Primary system entity working in a virtual Silo
Virtual Silo Flavour	virtual silo type, e.g. a "Mobius flavour" is related to a virtual silo with Mobius broker, a "MQTT flavour" refers to a virtual silo with MQTT broker, etc.
Virtual Thing (vThing)	An emulation of a real thing that produces data obtained by processing/controlling data coming from real things.
Tenant	User that access the Fed4IoT VirIoT platform to develop IoT applications through a vSilo
Root Data Domain	Set of sources providing IoT information to the VirIoT platform.
Federated system	External IoT systems that share information with VirIoT (through the System vSilo) such as to form a NGSI-LD global federated system.
System vSilo	NGSI-LD vSilo used at system level to share information of vThings with external NGSI-LD federated systems.

Table 2: Fed4IoT Dictionary

1 Introduction

1.1 Purpose of the Document

Deliverable D4.1 focuses on technologies and models used to share information among the different cross-domain actors of the Fed4IoT architecture that we described in deliverable D2.2. Specifically, the information sharing solutions hereafter discussed are used to:

- internally share information among the VirIoT components (internal information sharing);
- externally share VirIoT generated information (by ThingVisors) with other federated IoT platforms (external information sharing);
- carry out end-to-end cross-domain data model translations, starting from the heterogeneous data models used in the Root Data Domain, passing through the NGSI-LD neutral format internally used in the VirIoT data plane, eventually arriving to the data model used in vSilos' brokers.

The deliverable is structured in three main sections covering the above topics, plus one that carries out a performance evaluation of a critical architectural component, i.e. the internal MQTT dispatching system.

Section 2 describes internal and external information sharing services. The section initially provides a revision of the system architecture, identifying technologies and data models used on the different interfaces. Since many NGSI-LD components [1] (that are under active specification at the ETSI CIM Industry Specification Group) are used to implement our information sharing services, the section provides an overview of the NGSI-LD standard, mainly focusing on those parts where Fed4IoT is contributing, namely the data model, the API and the architecture. Indeed, the usage of NGSI-LD within a highly scalable environment, as focused by Fed4IoT, requires us to pay specific attention to different NGSI-LD aspects including:

- the NGSI-LD data model (see section 2.1.1), which should be able to easily support the aforementioned cross-domain translations;
- the federated architecture of NGSI-LD (see section 2.1.2) and related API, which should be able both to create a large scale federation of IoT information systems where the VirIoT platform is one of them, and to efficiently scale-up and make distributed specific NGSI-LD components that are internally used in the VirIoT architecture, such as the Registry Servers and the System vSilo, hereafter described.

Besides, Section 2 describes the new architectural components we introduced to support the information sharing services, i.e. the Semantic Discovery Component (for vThings discovery) and the the System vSilo (for external information sharing). Then, we give details of our pub/sub topic-based control and data plane used for internal information sharing, and how we investigated, in section 4, its capability to up-scale to

high-capacity service levels. Eventually, we describe a possible use of Information Centric Networkig (ICN) as an information sharing technology to implement distributed ThingVisors.

In Section 3 we describe and recommend strategies to carry out some end-to-end cross-domain data model translations by having NGSI-LD as middle/neutral format. Thus, we considered the following mappings, where NGSI-LD is an end-point: (a) SenML–NGSI-LD, (b) NGSIv2–NGSI-LD, (c) oneM2M–NGSI-LD. These information models are often not only domain specific but also tailored to specific needs of their consumers. This implies that it is often very difficult to achieve a semantic mapping through a generic processor, even in the case of expressive models (i.e. expressed through RDFS ontologies). The section discusses these aspects, starting with a simple case (SenML) and concluding with more a complicated one, i.e. with oneM2M by leveraging SmartM2M SAREF ontologies.

Finally, in section 5, we describe the security aspects focused by the project up to now that concern: i) Authentication, Identity Management and Access Control solutions used by users to access the VirIoT platform; ii) Data Centric security used to secure the information exchange within VirIoT and in the external federation; iii) the use of a Distributed Ledger Technology to track the sharing of information between VirIoT and the source of information in the Root Data Domain.

1.2 Executive Summary

This deliverable is the first deliverable of the D4.x deliverable series, introducing initial results coming from Tasks 4.1, 4.2 and 4.3.

Task 4.1, Information model, is devoted to set up a generic and extensible model that covers the information from different domains to facilitate the federation at data level. Task 4.2, Information sharing federated architecture, is devoted to set up federated information sharing solutions for multi-domain information interoperability. Task 4.3, Secure and efficient information dissemination, is devoted to design specific optimizations for the secure dissemination of information withing the Fed4IoT architecture.

Results in chapter 2 concern an update to the Fed4IoT architecture, which was already introduced in deliverable D2.2. We introduce new, specific components taking care of the Information Sharing at various levels (T4.2). We deal with semantic discovery of information and with sharing of information to external platforms in a federated fashion. We also focus on internal information sharing among the VirIoT's platform components, going into the details of our pub/sub-based control and data planes, and we introduce preliminary ideas on how to use ICN-centric approach to deploy distributed ThingVisors (T4.3), i.e. to implement a "ThingVisor Factory". The issue of massively scaling-up a pub/sub topic-based info distribution architecture in analysed, by means of models and simulation, with great detail in section 4 MQTT Clustering Technologies (T4.3).

Results in section 3 concern the mapping from/to various different heterogeneous information models, into the reference NGSI-LD model (T4.1).

Results in section 5 concern the security technologies we explored, adopted and

adapted in our architecture (T4.2).

1.3 Quality Review

The internal Reviewer responsible of this deliverable is Andrea Detti (CNIT).

Version Control Table			
V.	**Purpose/Changes**	**Authors**	**Date**
0.1	ToC	Giuseppe Tropea (CNIT)	05/11/2019
0.2	ToC revision, guidelines and introduction	Franck LE Gall (EGM)	05/12/2019
0.3	Initial NGSI-LD related content in chapter 2	Benoit Orihuela (EGM)	21/12/2019
0.4	Chapter 2 Information Sharing Architecture	Martin Bauer (NEC), Andrea Detti, Giuseppe Tropea (CNIT), Hidenori Nakazato, Kenji Kanai (WAS)	22/12/2019
0.5	Chapter 5 Security	Juan A. Martinez, Antonio Skarmeta (OdinS), Giuseppe Tropea (CNIT), Hamza Baqa (EGM), Kenichi Nakamura (PAN)	23/12/2019
0.6	Chapter 3 Cross-Doman Information Model	Ahmed Abid, Franck Le Gall (EGM), Juan A. Sanchez (OdinS)	24/12/2019
0.7	Chapter 4 MQTT Clustering Technologies	Andrea Detti, Ludovico Funari (CNIT)	27/12/2019
0.8	Overall cleanup and harmonization	Gisueppe Tropea (CNIT)	28/12/2019
0.9	Final review	Andrea Detti (CNIT)	30/12/2019

Table 3: Version Control Table

2 Information Sharing Services

By Fed4IoT Information Sharing Services we refer to the technologies and data models used in the VirIoT platform to exchange information on the different interfaces.

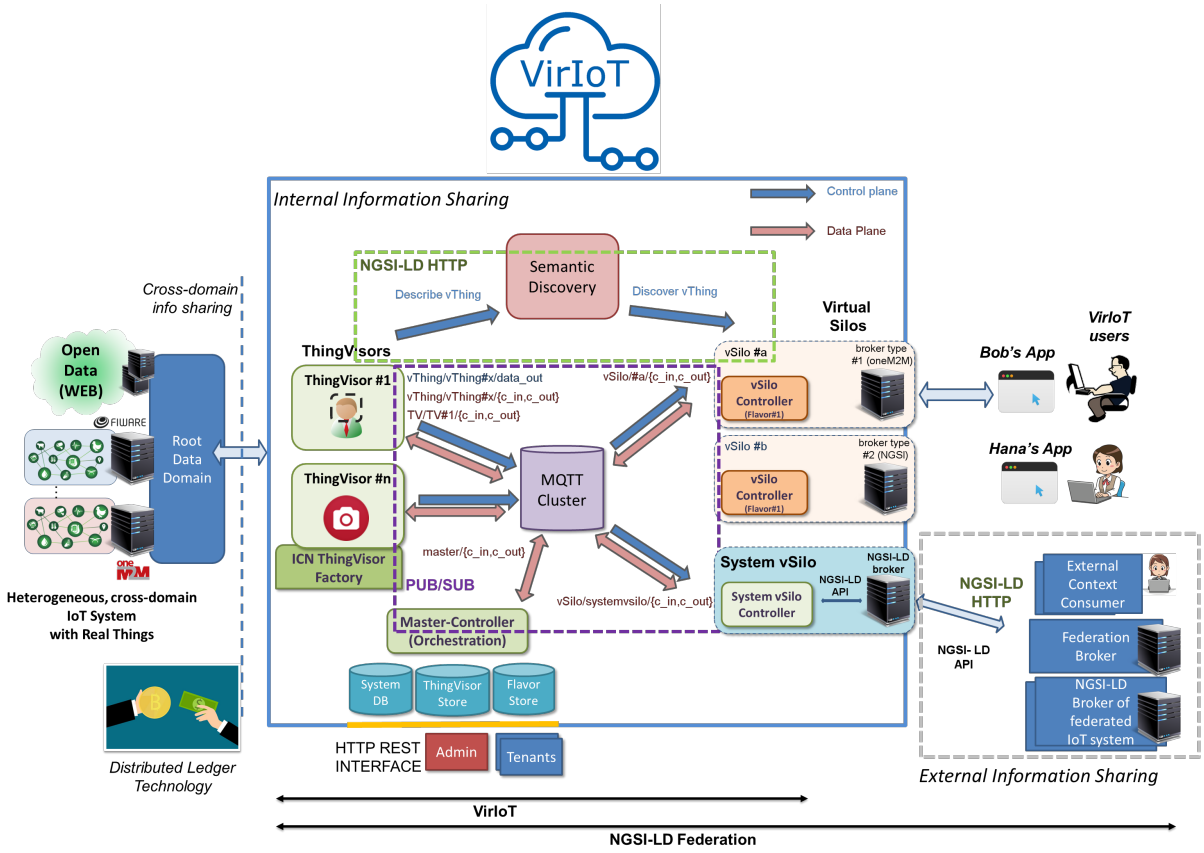

Figure 1: Fed4IoT architecture including the new System vSilo and Semantic Discovery components.

Figure 1 shows the VirIoT architecture, which is an evolution of what we presented in D2.2, and it highlights the different solutions (NGSI-LD HTTP, PUB/SUB, ICN, etc.) used to share information on the different interfaces. These solutions are extensively described in this deliverable.

Firstly, we introduce the two extensions we made to the architecture presented in D2.2, because they have a key role in the information sharing services:

- We considered the possibility of **federating the VirIoT platform with external IoT platforms**, thus forming a federation of cross-domain IoT data, where VirIoT contributes information through its vThings. The federation makes it possible to access vThing data not only for users of VirIoT, within their vSilos, but also for users of other federated IoT systems (external context consumers). For the purpose of federation, we promote NGSI-LD as the inter-working standard, by creating an

NGSI-LD entry-point to the data of our VirIoT platform and by fully supporting the NGSI-LD API through a **System vSilo**. All newly created vThings are, by default, added to the System vSilo, regardless of any other user-owned vSilos. The System vSilo is based on a NGSI-LD internal broker, so that it is able to export data from its vThings into NGSI-LD, serving the purpose of an harmonized connector to external federated platforms. Hence, we allow the VirIoT platform to interconnect with an external federation of NGSI-LD brokers and an eco-system of cross-cutting context information consumers and producers.

- We introduced a **Semantic Discovery** functionality which is used by platform users (or tenants) to (1) find out what vThings are currently available within the VirIoT platform, that match specific semantic criteria, in order to add them to their vSilos or (2) be notified when new vThings appear, that match the set of criteria. In this case, too, we exploited NGSI-LD components, specifically the Registry Server, and we defined the interfaces to register and find available data sources based on descriptors and semantic typing.

In terms of scope of the sharing, Fed4IoT's Information Sharing Services are organized along the following macro-areas, depicted in Figure 1.

Internal Information Sharing – whenever we have to design means to pass information among components of the VirIoT platform. Internal information concerns both control and data planes. VirIoT control commands (see D3.x) travel in the control plane, for instance for switching on and off platform components (vSilos, ThingVisors, etc.), or configuring them. Data flows, encapsulating vThings' data items that need to be carried among platform components (e.g. from ThingVisors to vSilos), travel in the data plane. The data model used in the data plane comes from NGSI-LD, which is used internally as a "neutral format" able to capture the structure and semantics of all other data formats used within the Root Data Domain (see D2.2, chapter 6). These NGSI-LD data pieces are transferred using MQTT pub/sub services, based on a cluster of MQTT brokers for scaling purposes. Most of the information of the control plane uses these MQTT pub/sub services too (see Section 2.4), but the newly introduced Semantic Discovery component currently uses the NGSI-LD HTTP API and data model, based on the NGSI-LD Registry Server (see Section 2.2). We also start to introduce the ThingVisor Factory concept, and we position it within the Fed4IoT architecture. A ThingVisor Factory is actually a tool for building distributed ThingVisors, thus exploiting the edge/core flexible computing virtualization services described in D3.x. Accordingly, an ICN-based request/response information sharing approach for function chaining is used to implement an ICN-based ThingVisor Factory (see Section 2.5) [1].

External Information Sharing – whenever we have to share information between VirIoT and external federated IoT systems. To this aim we can use the System vSilo, which is based on a NGSI-LD broker. The System vSilo is part of a larger NGSI-LD architecture formed by a Federated Broker, other NGSI-LD brokers, etc., as discussed in Section 2.3

[1] We remind that other solutions (e.g. FogFlow) can be used to build distributed ThingVisors.

In what follows we present the NGSI-LD standard, mainly focusing on those parts where Fed4IoT is contributing to the standardization process because they are used in the architecture, as previously mentioned. Then we present the new architectural components, i.e. the System vSilo and the Semantic Discovery Component. Later we present the pub/sub topic-based internal information sharing describing topics and messages. Lastly, we present the sharing of information within the ICN based ThingVisor Factory.

Then, subsequent sections of the deliverable cope with: i) the cross-domain information sharing taking place between the Root Data Domain and the VirIoT platform (see Figure 1); ii) a scalability analysis of the MQTT cluster that supports internal information sharing services; iii) security aspects, including distributed ledger technologies to be used to track sharing of information between the Root Data Domain and the VirIoT platform, e.g. for business purposes such as selling data.

2.1 The NGSI-LD standard from ETSI CIM ISG

The cross-domain Context Information Management (CIM) is a standardization initiative from ETSI, in the form of an Industry Specification Group (ISG). It is a group of about 22 ETSI members and 7 non-ETSI organisations that agreed to co-define the core NGSI-LD specification and many related documents.

The goal of ISG CIM is to develop interoperable software implementations of a cross-cutting solution for managing context information, named NGSI-LD.

Currently, the following Fed4IoT partners are CIM members/participants:

- NEC
- EGM
- OdinS
- CNIT

Especially relevant to Fed4IoT's Information Sharing Services is the fact that the group expects data is re-usable without the applications being part of some specific "vertical", integrating across administrative, technological and sometimes also language boundaries. Having a model for context information ("data about data") such as who generated it, when, how, what accuracy, what license, etc., is especially relevant for a virtualization platform acting as de-coupling intermediary between the sensors' infrastructure (which is producing data) and the data brokering to end-users and applications.

There are two facets of NGSI-LD: (a) the NGSI-LD API which is a means of exchanging Context Information, and (b) the NGSI-LD Information Model which is a meta model on which the NGSI-LD API functionality is based and which is defined as a high-level ontology to enable interoperabilty of terms/concepts across vertical domains.

Fed4IoT is actively contributing to both.

2.1.1 NGSI-LD Information Model and API

Fed4IoT has agreed, since the first period of the project, NGSI-LD to be the reference data model of the project, both for modelling and querying data. We re-assess in this section the main concepts of NGSI-LD information model and API.

NGSI-LD is represented in JSON-LD and, consequently, has a grounding in RDF. This grounding allows it to capture high-level relationships between entities (i.e. IoT devices, group of devices or non-IoT information) and properties of entities, as shown below. The core concept in the NGSI-LD data model is the "Entity", which can have properties and relationships to other entities. An entity is equivalent to an Owl class. The assumption is that the world consists of entities (please refer to ETSI CIM ISG WorkItem 006, Information Model, available online at the group's open area), that usually are physical entities like a car or a building, but can be more abstract things, like a company or the coverage area of a WLAN access point, too.

Entity instances are identified by a unique URI and a type, e.g. a sensor with identifier `urn:ngsi-ld:Sensor:01` and of type `Sensor`. Different from rdf:Properties, NGSI-LD properties (and relationship) are also considered as Owl classes, too. Properties and relationships can be annotated by properties and relationships themselves, e.g. a timestamp, the provenance of the information or the quality of the information can be provided. The hasObject and hasValue in the NGSI-LD meta-model are defined to enable RDF reification, based on the blank node pattern, so as to leverage the property graph model.

The NGSI-LD cross-domain ontology extends the NGSI-LD meta-model to cover several general contexts presented below [2]:

- Mobility defines the stationary, movable or mobile characteristics of an entity;
- Location differentiates and provide concepts to model the coordination based, set based or graph-based location;
- Temporal specification includes property and values for temporal property definitions;
- Behavioural system includes properties and values to describe system state, measurement and reliability;
- System composition and grouping provides a way to model system of systems in which small systems are composed together to form a complex system following specific patterns.

The NGSI-LD cross domain ontology is presented in Figure 2:

Based on this high-level ontology, the NGSI-LD API supports several operations. The API is the standard for management of context information (which can be summarised has being any piece of information associated to a context such as time-location information). The Overall NGSI-LD API operations include:

- General Operations: which include creating, updating, retrieving, deleting entities and subscribing to NGSI-LD entities,

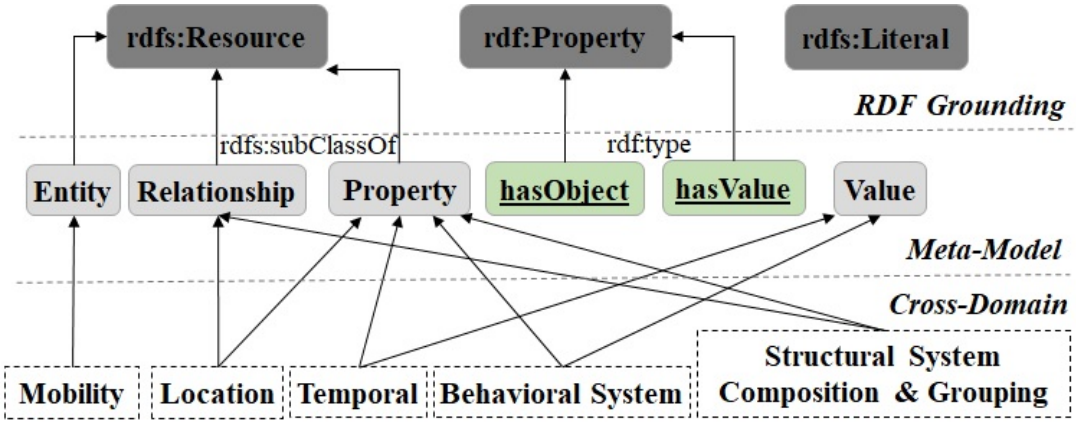

Figure 2: RDF standards to capture high-level relations between entities in NGSI-LD.

- Registry Operations: which include Context Source Registration operations on creating, updating, deleting, retrieving and subscribing to NGSI-LD entities,
- Batch Operations: which include operations of creating, updating and deleting NGSI-LD batch entities,
- Temporal Operations: which include operations of creating, updating (adding and modifying attributes) of NGSI-LD temporal entity representation.

2.1.2 NGSI-LD architecture

NGSI-LD as specified in ETSI ISG CIM primarily defines an API and an information model, not a single architecture. Nevertheless, it has been the intention to enable different architectures based on NGSI-LD, in particular centralized, distributed and federated architectures. In this section we introduce the general architectural assumptions behind NGSI-LD and then show how these enable the implementation of different functionalities and components in Fed4IoT.

Figure 3 introduces components in the sense of roles and their respective interactions. Context Consumers request context information via the NGSI-LD API, either synchronously using queries or asynchronously using subscriptions, which result in notifications.

Context is originally provided by Context Producers and Context Sources. Whereas Context Producers use the management methods of the NGSI-LD API to create, update or delete context information in a Broker component, Context Sources store the information themselves and offer the full interface for querying or subscribing to the information. Context Sources register what kind of information they have with the Registry Server.

Context Consumers either request context information from Broker or directly from Context Sources. To find the relevant Context Sources, Brokers or Context Consumers discover the relevant Context Sources via the Registry Server based on the registrations. Brokers then request the information from the identified Context Sources, aggregate it

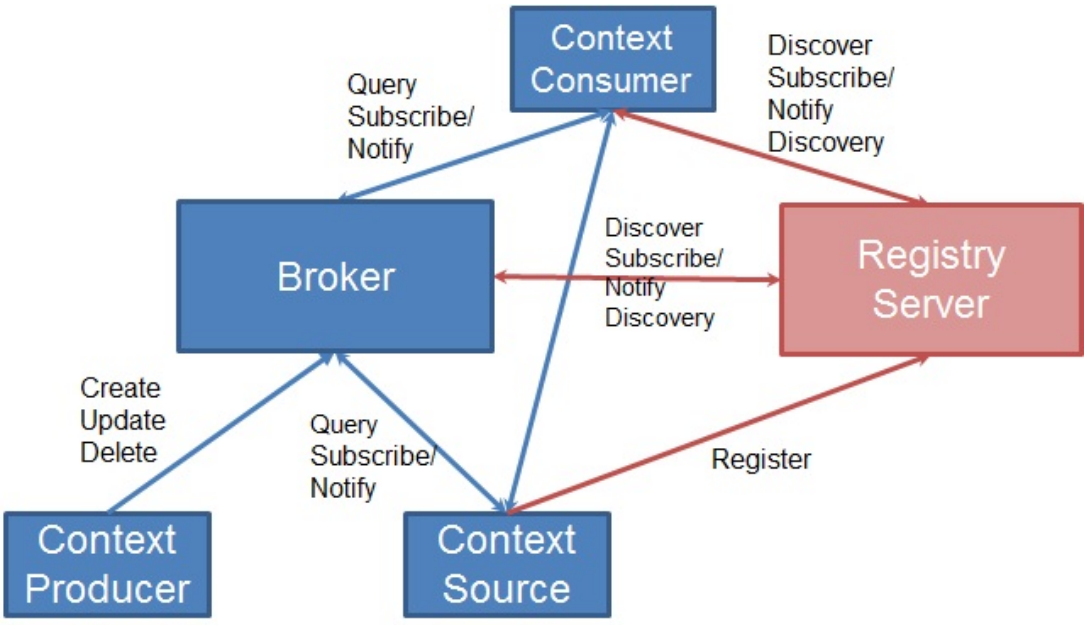

Figure 3: Roles and interactions in NGSI-LD architecture.

and provide it to Context Consumers. Brokers in addition act as Context Sources for the information in their local storage, which was provided by Context Producers.

2.2 Semantic Discovery Component

This section describes the Semantic Discovery component as depicted in the information sharing architecture in Figure 1 and shows how it can be implemented using Scorpio's Registry Server based on the NGSI-LD API.

The role of the Semantic Discovery in the Information Sharing Architecture is to enable the semantic discovery of vThings. This functionality is relevant for setting up a vSilo, i.e. for finding the relevant vThings to subscribe to in order to populate the vSilo with information. To enable the discovery, ThingVisors have to register the meta information about their vThings, i.e. which entities, entity types, properties and relationships are available, for which geographic area they can provide information, provenance information like the source of the information, the quality of the information etc.

The Semantic Discovery in the Information Sharing Architecture can be implemented using the Registry Server as defined in NGSI-LD, in this case the Registry Server implementation of the Scorpio NGSI-LD Broker. The Context Source Registration information element can be used to describe a vThing as it can contain multiple entity descriptions, i.e. entity id (optional), entity type and properties (optional), relationships (optional) as well as geographic location specified in GeoJSON and optionally user-defined attributes containing provenance information like the source of information or the quality of information, thus fulfilling the requirements for the Semantic Discovery.

The Registry Server in NGSI-LD typically enables Consumers and Brokers to discover

relevant Context Sources providing the URL of the access point implementing the NGSI-LD API.

For this purpose we can use the NGSI-LD Registry Server, which provides an HTTP-based request/response interface as part of the standardized NGSI-LD API. Context Sources register themselves with the information they can provide and the URL of their access point. In general, different granularity of registrations is possible, e.g. entity id, entity type and attributes or just entity types and attribute or even only entity type are possible, resulting in different trade-offs regarding the load of registration updates and the query load. ThingVisors provide information about one or more entities, i.e. they would typically register with a number of entity id and entity type pairs. Instead of registering the URL of their access point, they register the vThing to which consuming Virtual Silos have to subscribe to in order to receive the entity information. This is encoded by prefixing "vThing:" to the name of the vThing as shown in the example below.

***Describe vThing* implemented using NGSI-LD Context Source Registrations.**

```
{
 "id":"urn:ngsi-ld:ContextSourceRegistration:santander:parkingspots",
 "type":"ContextSourceRegistration",
 "name":"Parking Spots and Weather Observed Santander Registration",
 "description":"This registration registers the Parking Spots in
    Santander, some of which are also equipped with temperature sensors
    exposed via a WeatherObserved entity.",
 "information": [
 {
  "entities": [
   {
    "id": "urn:ngsi-ld:ParkingSpot:santander:3921",
    "type": "https://uri.fiware.org/ParkingSpot"
   },
   {
    "id": "urn:ngsi-ld:ParkingSpot:santander:3922",
    "type": "https://uri.fiware.org/ParkingSpot"
   },
   {
    "type": "https://uri.fiware.org/WeatherObserved"
   }]
 }],
 "endpoint":"vThing:urn:ngsi-ld:ParkingSpot:santander:parkingspots",
 "location": {
  "type":"Polygon",
  "coordinates": [[
   [-3.885801, 43.502139],
   [-3.735709, 43.502139],
   [-3.735709, 43.414834],
   [-3.885801, 43.414834],
   [-3.885801, 43.502139]
  ]]
 }
 "origin":"Santander Car Park Management",
 "timestamp": {"start": "2017-06-12T13:53:15Z"},
 "expires": "2030-06-12T13:53:15Z"
}
```

The above JSON registration gives an example, where two parking spots are explicitly registered, providing both entity id and type, and in addition entities of type WeatherObserved are registered, which provide the Property temperature. Furthermore, the location of the parking spots is specified using GeoJSON and the provenance information ”origin”

is given as "Santander Car Park Management".

For the discovery, the requester, i.e. the one setting up the vSilo, has to specify in which entities it is interested and filter according to further aspects, e.g. location and origin. Regarding entities, either specific entities can be selected by providing entity id and type or all entities of a given type can be selected. On this set of entities, the geographic filter and the filter on other attributes like origin is applied, resulting in the reduced set of fitting vThings, which are returned and to which the vSilo then subscribes.

The discovery can be mapped to NGSI-LD, which provides the relevant functionality. An example discovery request for discovering the property temperature as part of WeatherObserved entities in Santander, specified as a geographic coordinate and a radius of 3000m around it, is shown below.

***Discovery* of WeatherObserved entities with porperty temperature.**

```
GET /ngsi-ld/v1/csourceRegistrations?type=WeatherObserved&amp;
attrs=temperature&amp; geoproperty=location&amp;
georel=near;maxDistance==3000&amp; geometry=Point&amp;
coordinates=%5B-3.804109%2C43.464704%5D HTTP/1.1
Accept: application/json
Link: <https://pastebin.com/raw/rQ9mU4ue>;
rel="http://www.w3.org/ns/json-ld#context"; type="application/ld+json"
Host: <hostname>:2042
```

In case only vThings provided by `Santander Car Park Management` are to be considered, an additional filter `csf=origin=="Santander Car Park Management"` can be added.

2.3 System vSilo

The System vSilo's goal, as presented in Section 2, is to share a subset of the available information with the external world. In Fed4IoT such a vSilo would be implemented based on the Scorpio Broker implementation (by NEC). In the simplest case, there could be one Scorpio Broker instance that would make information available to the external world, via the NGSI-LD API using its HTTP binding. Such a Broker can be part of an external Federation and for this purpose the available information in the Broker can be registered with an external Registry server. The registration can either automatically be done on the chosen granularity level, e.g. a detailed registration of every NGSI-LD entity available or on an entity type level. Alternatively, a tailor-made registration can be manually registered, e.g. exposing only a subset of the entity information actually available in the Broker.

Figure 4 shows an example of a federation, where the System vSilo and an External Broker are part of a federation. Both are registered with the Registry Server on the federation level. On request from a Context Consumer, the Federation Broker will check for potential sources in the Registry Server and forward the request to the System vSilo

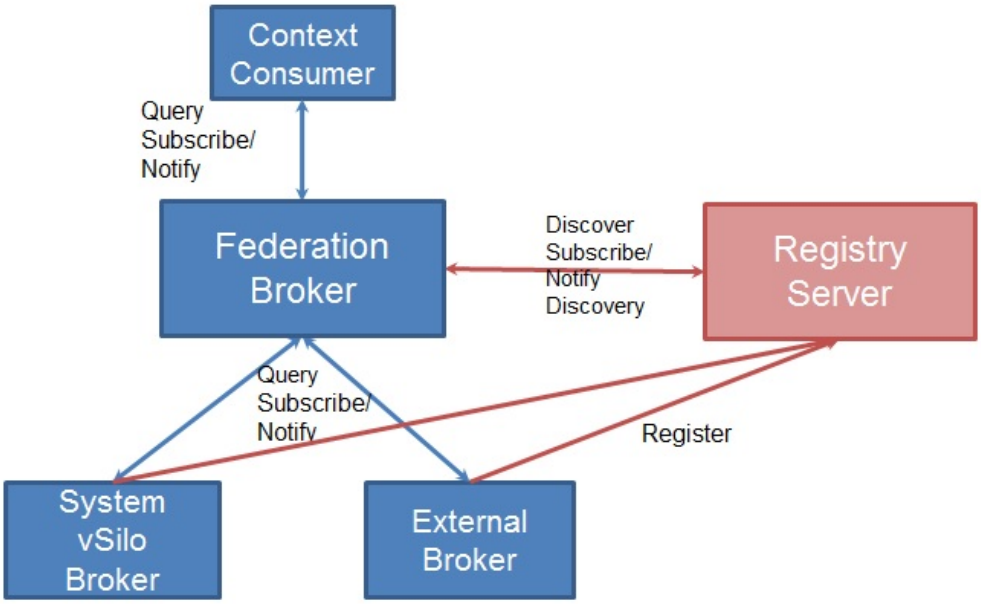

Figure 4: External NGSI-LD Federation.

Broker, the External Broker or both, in case both may have relevant information. In the latter case, the returned information will be aggregated before returning it to the Context Consumer.

2.3.1 HTTP-based Information Sharing to External Platforms

The NGSI-LD API specification defines an exhaustive HTTP binding with a RESTful API. This HTTP binding can be divided in six main categories:

- Context information provision
- Context information consumption
- Context information subscription
- Context source registration
- Context source discovery
- Context source registration subscription

In the context of the System vSilo, some categories are extensively used, as summarized in Table 4.

- Context information provision: it is used by the System vSilo Controller to create and keep up to date the information context relative to the virtual things the System vSilo is managing.

- Context information consumption: it is used by the external platform to discover entities belonging to VirIoT.
- Context information subscription: it is used mainly by external platform to subscribe and be automatically notified, in a reactive way, of any event it is interested in.

Category	Actor	Purpose
Context information provision	System vSilo Controller	Create information and metadata about the vThings into the NGSI-LD broker of the System vSilo
Context information provision	System vSilo Controller	Enrich information context into the NGSI-LD broker of the System vSilo with data produced during the vSilo usage
Context information consumption	System vSilo Controller	Search and discover information about entities known to the VirIoT platform by contacting the Semantic Discovery component
Context information consumption	External platforms	Search and discover (authorized) information about entities known to the VirIoT platform
Context information subscription	External platforms	Notify about changes and events on specific context information

Table 4: Summary of NGSI-LD HTTP API usages in the System vSilo

2.3.1.1 Context Information Provision

The primary objective of the context information provision HTTP API is to create the information context and to keep it up to date. Specifically, it is used to provision entities (e.g. vThings and their real counterparts) enriched with their metadata (e.g. location, observation space, ...), and to keep such information up to date over time, whether it is vThing direct information or metadata (e.g. "update the position of this vThing", "update the current value of a temperature sensor", ...) or vThing more global context by managing its relations and properties (e.g. "add a humidity temporal property to this vThing" or "add a relationship between these two vThings").

To explain it in a more detailed way, some of the API usage are described below with concrete examples.

First, here is an example call invoked when a vThing wishes to provision itself:

POST /ngsi-ld/v1/entities


```
{
    "id": "urn:ngsi-ld:ParkingSpot:SantanderCenter:3",
    "type": "ParkingSpot",
    "status": {
        "type": "Property",
        "value": "free",
        "observedAt": "2019-12-02T12:00:00Z"
    },
    "category": {
        "type": "Property",
        "value": [
            "onstreet"
        ]
    },
    "refParkingSite": {
        "type": "Relationship",
        "object": "urn:ngsi-ld:ParkingSite:santander:SantanderCenter"
    },
    "name": {
        "type": "Property",
        "value": "A-13"
    },
    "location": {
        "type": "GeoProperty",
        "value": {
            "type": "Point",
            "coordinates": [
                -3.80356167695194,
                43.46296641666926
            ]
        }
    },
    "@context": [
        "https://ontology.fed4iot.org/context/fed4iot.jsonld",
        "https://uri.etsi.org/ngsi-ld/v1/ngsi-ld-core-context.jsonld"
    ]
}
```


HEADERS:

Accept: application/json
Content-Type: application/ld+json

RESPONSE: HTTP/1.1 201 Created

```
(Empty response body)
```

HEADERS:

Location: /ngsi-ld/v1/entities/urn:ngsi-ld:ParkingSpot:SantanderCenter:3

Then, let's see an example call invoked when a vThing wishes to update the value of one of its properties:

PATCH /ngsi-ld/v1/entities/urn:ngsi-ld:ParkingSpot:SantanderCenter:3/attrs/name

```
{
  "name": "A-14"
}
```

HEADERS:

Link: <https://ontology.fed4iot.org/context/fed4iot.jsonld>; rel=http://www.w3.org/ns/json-ld#context; type=application/ld+json
Accept: application/json
Content-Type: application/json

RESPONSE: HTTP/1.1 204 No Content

```
(Empty response body)
```

Similarly, when a vThing evolves and is enriched with a new property, it propagates this event by calling the following API:

POST /ngsi-ld/v1/entities/urn:ngsi-ld:ParkingSpot:SantanderCenter:3/attrs

```
{
  "occupiedBy": {
    "type": "Property",
    "value": "Toyota Yaris"
  }
}
```

HEADERS:

Link: <https://ontology.fed4iot.org/context/fed4iot.jsonld>; rel=http://www.w3.org/ns/json-ld#context; type=application/ld+json
Accept: application/json
Content-Type: application/json

RESPONSE: HTTP/1.1 204 No Content

```
(Empty response body)
```

2.3.1.2 Context Information Consumption

The primary objective of the context information consumption is to let others (vSilos and third parties in the VirIoT platform) perform searches and discover interesting entities. For that, we use a range of the powerful NGSI-LD HTTP API search capabilities, from simple basic ones (e.g. "give me all the entities of type Parking Spot" or "give me all the entities that provide information about temperature") to more complex ones including time and space based filters (e.g. "give me all the entities that are located in this given area" or "give me all the parking spots whose status is busy since more than 5 days").

To explain it in a more detailed way, some of the API usage are described below with concrete examples.

First, here is an example call invoked when someones wants to retrieve all known information about a given entity:

GET /ngsi-ld/v1/entities/urn:ngsi-ld:ParkingSpot:SantanderCenter:3

```
(Empty request body)
```

HEADERS:

Link: <https://ontology.fed4iot.org/context/fed4iot.jsonld>; rel=http://www.w3.org/ns/json-ld#context; type=application/ld+json
Accept: application/ld+json
Content-Type: application/json

RESPONSE: HTTP/1.1 200 OK


```
{
    "id": "urn:ngsi-ld:ParkingSpot:SantanderCenter:3",
    "type": "ParkingSpot",
    "status": {
        "type": "Property",
        "value": "free",
        "observedAt": "2019-12-02T12:00:00Z"
    },
    "category": {
        "type": "Property",
        "value": [
            "onstreet"
        ]
    },
    "refParkingSite": {
        "type": "Relationship",
        "object": "urn:ngsi-ld:ParkingSite:santander:SantanderCenter"
    },
    "name": {
        "type": "Property",
        "value": "A-13"
    },
    "location": {
        "type": "GeoProperty",
        "value": {
            "type": "Point",
            "coordinates": [
                -3.80356167695194,
                43.46296641666926
            ]
        }
    },
    "@context": [
        "https://ontology.fed4iot.org/context/fed4iot.jsonld",
        "https://uri.etsi.org/ngsi-ld/v1/ngsi-ld-core-context.jsonld"
    ]
}
```


Then, let's say the same consumer now wants to get all information about parking spots located in the SantanderCenter parking site:

GET /ngsi-ld/v1/entities?type=urn:ngsi-ld:ParkingSite &q=refParkingSite==urn:ngsi-ld:ParkingSite:SantanderCenter

```
(Empty request body)
```

HEADERS:

Link: <https://ontology.fed4iot.org/context/fed4iot.jsonld>; rel=http://www.w3.org/ns/json-ld#context; type=application/ld+json
Accept: application/json
Content-Type: application/json

RESPONSE: HTTP/1.1 200 OK

```
(A list of parking stops as shown in the previous example)
```

2.3.1.3 Context Information Subscription

The primary objective of the context information subscription is to allow internal and external consumers to subscribe to changes in the information context. A change can virtually be anything: an updated value for a property, a new entity, a new entity in a specific area, etc.

Then, once a subscriber has registered its subscription, it is automatically notified as soon as an event triggers the subscription.

This is of particular interest for third parties that can have a reactive notification and thus do not need to be constantly polling the registry in case there is something new. It is also used by vSilos to extend their knowledge and receive updated information beyond their vThings for which they already receive notifications about via a specific MQTT topic.

First, here is an example call invoked when someone wants to register a subscription. Here it registers for notifications about changes in the "status" property for the `urn:ngsi-ld:ParkingSpot:SantanderCenter:3` parking spot. When the "status" property changes, it will receive the new value for this property but also the value of the "occupiedBy" one.

POST /ngsi-ld/v1/subscriptions

```
{
  "type": "Subscription",
  "entities": [{
        "idPattern": "urn:ngsi-ld:ParkingSpot:SantanderCenter:3",
        "type": "ParkingSpot"
  }],
  "watchedAttributes": ["status"],
  "notification": {
        "attributes": ["status", "occupiedBy"],
        "format": "keyValues",
        "endpoint": {
               "uri": "http://my.endpoint.org/notify",
               "accept": "application/json"
        }
  }
}
```

HEADERS:

Link: <https://ontology.fed4iot.org/context/fed4iot.jsonld>;

rel=http://www.w3.org/ns/json-ld#context;

type=application/ld+json

Accept: application/ld+json

RESPONSE: HTTP/1.1 201 Created

```
(Empty response body)
```

HEADERS:

Location: /ngsi-ld/v1/subscriptions/urn:ngsi-ld:Subscription:0123

Of course, it also allows for more fine-grained subscriptions. In the following example, it restricts notifications on the same "status" property to cases where the parking spot is not occupied by Tesla cars:

POST /ngsi-ld/v1/subscriptions

```
{
  "type": "Subscription",
  "entities": [{
        "idPattern": "urn:ngsi-ld:ParkingSpot:SantanderCenter:3",
        "type": "ParkingSpot"
  }],
  "watchedAttributes": ["status"],
  "q":"https://ontology.fed4iot.org/context/fed4iot.jsonld#occupiedBy!=
     Tesla",
  "notification": {
        "attributes": ["brandName", "occupiedBy"],
        "format": "keyValues",
        "endpoint": {
               "uri": "http://my.endpoint.org/notify",
               "accept": "application/json"
        }
  }
}
```

The subscription owner is then automatically notified each time an occurring event matches its subscription (e.g. when the status of the parking stop with id `urn:ngsi-ld:ParkingSpot:SantanderCenter:3` changes).

This is materialized by an HTTP request on the endpoint registered at the time the subscription was created (or updated):

POST "endpoint.uri"


```
{
    "id": "urn:ngsi-ld:ParkingSpot:SantanderCenter:3",
    "type": "ParkingSpot",
    "status": {
        "type": "Property",
        "value": "busy",
        "observedAt": "2019-12-10T12:00:00Z"
    },
    "refParkingSite": {
        "type": "Relationship",
        "object": "urn:ngsi-ld:ParkingSite:SantanderCenter"
    },
    "occupiedBy": {
        "type": "Property",
        "value": "Tesla model S"
    }
}
```


2.3.2 Distributed System vSilo

For scalability reasons, a single Broker may not be sufficient for implementing the System vSilo as the number of entities may be too large to handle. In this case, the System vSilo can be distributed according to suitable criteria. This could be based on the type of entity, the name of the vThing or the ThingVisor. The distribution strategy has to be applied by the System vSilo Controller when setting up the System vSilo. An example of a distributed setting based on the Scorpio NGSI-LD Broker.

Figure 5 shows an example of a distributed NGSI-LD-based System vSilo. The actual information is distributed into different Brokers according to the distribution strategy. These Brokers register themselves with the System vSilo Registry and all information can be accessed through the System vSilo Broker that accesses and aggregates information from the different Brokers based on the registration information in the System vSilo Registry.

2.4 Pub/Sub-based Internal Information Sharing

This section describes the use of a MQTT publish/subscribe distribution system for the internal information sharing of the VirIoT control commands and data items of Virtual Things specified in D3.1. Figure 6 shows the used topics. We have *control topics* and *data topics*. Control topics use `c_out, c_in` suffixes and data topics use `data_out, data_in` suffixes.

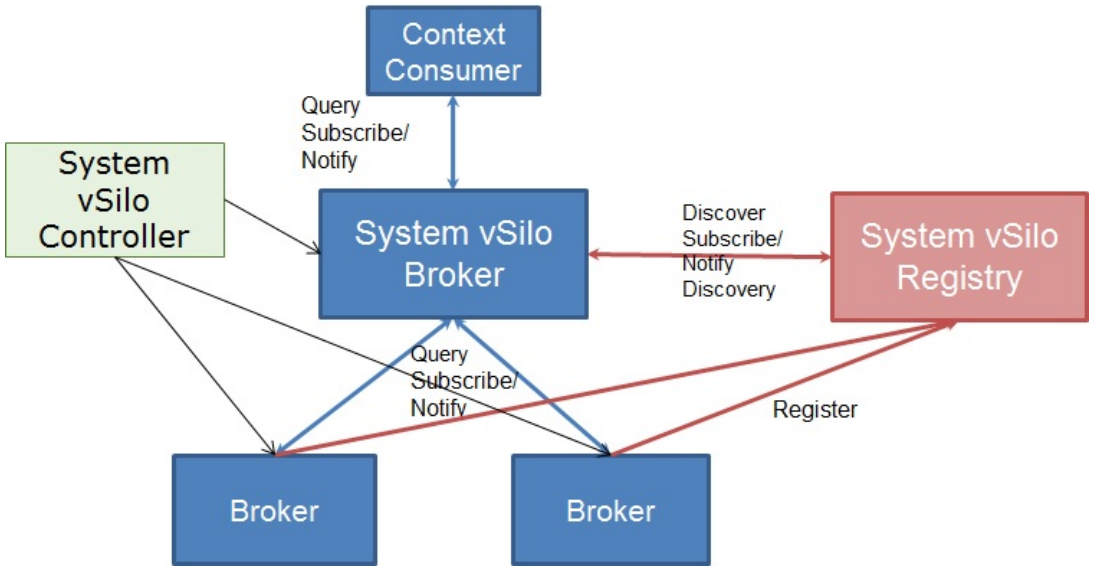

Figure 5: Distributed NGSI-LD System vSilo.

Figure 7 shows which control/data topics are used to transfer control commands and vThing data. Regarding the control topics, they start with a prefix that identifies the entity type (TV, for ThingVisor, vSilo for vSilo, etc.) followed by the unique ID of the related entity. Each entity (Master Controller, ThingVisor, vSilo controller, etc.) uses a specific input and output control topic to receive and send control messages, respectively.

A entity X is the only subscriber of its input control topic, other entities that wish to send information to the entity X publish on that input control topic. For instance, a vSilo whose ID is e.g. `tenant1/silo1` is subscriber of the topic `vSilo/tenant1/silo1/c_in`. When the Master Controller has to send it an `addVThing` control command, it publishes this message to the `vSilo/tenant1/silo1/c_in` topic.

A entity X is also the single publisher of its out topic. For instance, when a ThingVisor whose ID is e.g. `vWeatherTV`, wants to inform the rest of the system entities that it is going to be destroyed, it sends out a `deleteVThing` control messages for each vThing (e.g. `Rome/temp`) it handles on the vThing out control topic. e.g. `vThing/WeatherTV/Rome_temperature`. Each vSilo that is connected to this vThing is also a subscriber of this out control topic and thus will receive this control message and will remove the vThing entry from its internal broker.

Regarding the distribution of data produced by a vThing, the related ThingVisor publishes NGSI-LD data on the `vThing/<vThingID>/data_out` topic [2]. In case a vThing is associated to an actuator, it has to receive data coming from the vSilo controlling the actuator vThing (e.g. the change of state from OFF to ON). To this end, the ThingVisor is a subscriber of the topic `vThing/<vThingID/data_in` where this information can be pushed by vSilos.

[2] we remind that NGSI-LD is the neutral data format used to internally distribute information produced by a vThing

MQTT topics

Topic	Publisher	Subscribers	Description
		Control Topics	
vSilo/<vSiloID>/c_in	Master Controller ThingVisor	vSilo	Used to send control command to a specific vSilo
vSilo/<vSiloID>/c_out	vSilo	Master Controller	Used by a vSilo to send control command (e.g. destroy ack) related to the vSilo to interested entities (e.g. Master Controller)
TV/<thingVisorID>/c_in	Master Controller	ThingVisor	Used to send ThingVisor related control command to a specific ThingVisor
TV/<thingVisorID>/c_out	ThingVisor	Master Controller	Used by the ThingVisor to send ThingVisor related control command to interested entities (e.g. Master Controller)
vThing/<vThingID>/c_in	Master Controller vSilos	ThingVisor	Used to send control commands (e.g. getContextRequest) to the ThingVisor handling the specific vThing
vThing/<vThingID>/c_out	ThingVisor	Master Controller vSilos	Used by the ThingVisor to send vThing related control command to interested entities (e.g. vSilos connected to the vThing)
		Data Topics	
vThing/<vThingID>/data_out	ThingVisor of the vThing	vSilo controllers of vSilos connected to the vThing	NGSI-LD data items generated by a vThing
vThing/<vThingID>/data_in	vSilo controllers of vSilos connected to the vThing	ThingVisor of the vThing	NGSI-LD data items to be received by a vThing (e.g. used for actuators)

Figure 6: MQTT topics for internal information sharing

2.4.1 Performance of VirIoT's MQTT dissemination system

Currently, the MQTT broker is a critical part of VirIoT architectures, especially because any data generated by vThings cross it. In general, MQTT brokers can be deployed either as standalone servers or in a cluster configuration to increase reliability, availability and to increase the overall performances, as operations can be highly parallelized among the cluster nodes.

We have found out that the linear increases of the number of cluster nodes don't necessarily provide an equivalent linear gain in performance, and the penalty may be surprisingly significant. Accordingly, it is worth to understand the issue, provide simula-

VirIoT control commands - MQTT binding

Command	Publishers	Subscribers	Topic
addVThing	Master-controller	vSilo controller	vSilo/<vSiloID>/c_in
deleteVThing	Master-controller	vSilo controller	vSilo/<vSiloID>/c_in
createVThing	Thing Visor	Master-controller	TV/<thingVisorID>/c_out
destroyTV	Master-controller	Thing Visor	TV/<thingVisorID>/c_in
destroyTVAck	Thing Visor	Master Controller	TV/<thingVisorID>/c_out
destroyVSilo	Master-controller	vSilo controller	vSilo/<vSiloID>/c_in
destroyVSiloAck	Vsilo Controller	Master-controller	vSilo/<vSiloID>/c_out
deleteVThing	Thing Visor	vSilo controller	vThing/<vThingID>/c_out
getContextRequest	vSilo Controller	Thing Visor	vThing/<vThingID>/c_in
getContextResponse	Thing Visor	vSilo Controller	vSilo/<vSiloID>/c_in

VirIoT data items - MQTT binding

Content	Publisher	Subscribers	Topic
NGSI-LD data items generated by a vThing	ThingVisor of the vThing	vSilo controllers of vSilos connected to the vThing	vThing/<vThingID>/data_out
NGSI-LD data items to be received by a vThing (e.g. used for actuators)	vSilo controllers of vSilos connected to the vThing	ThingVisor of the vThing	vThing/<vThingID>/data_in

Figure 7: MQTT binding for internal data sharing

tion support for the modelling of the underlying phenomena and to eventually propose architectural approach to be followed in the final architecture release to cope with such a unscaling issue. This topic is addressed in-depth in chapter 4.

2.5 ICN-based Information Sharing for ThingVisor Factory

Information Centric Networking (ICN) can be used as an alternative to exchange messages among functions to compose ThingVisors, as explained in deliverable D3.1. Application of ICN is currently confined within implementation of ThingVisors (Figure 8). However, we

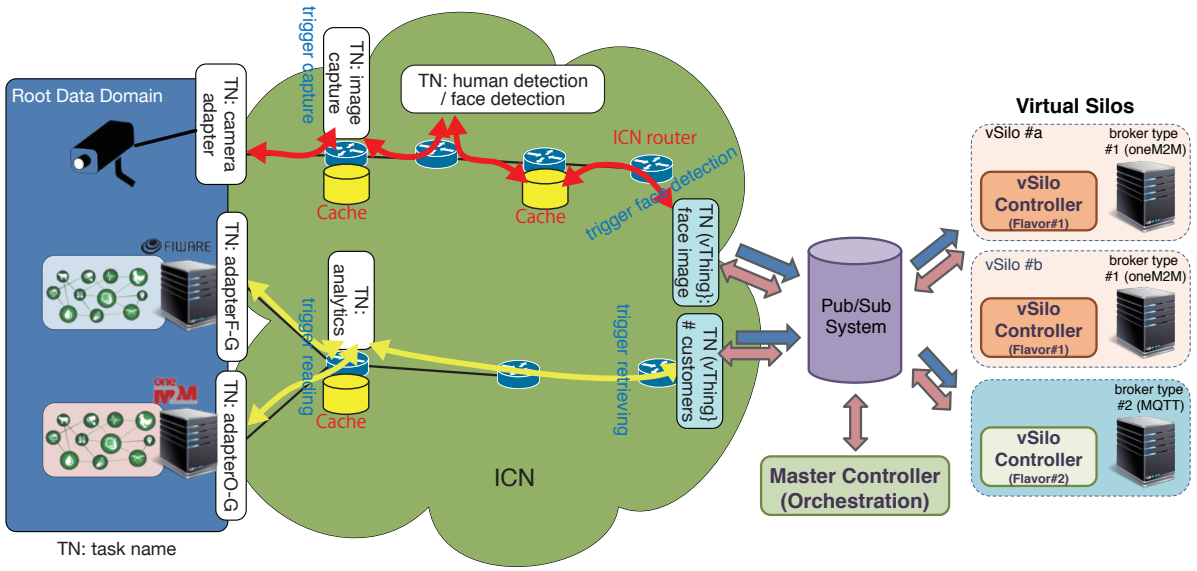

Figure 8: Application of ICN in Fed4IoT

are starting to design a concept of ThingVisor Factory, whereas the ICN will eventually be extended to co-exist with the the pub/sub system shown in Figure 8. We believe request/response communication model, as provided by ICN, is going to be a necessary addition to the publish/subscribe communication model which is currently provided in VirIoT.

In the current architecture as shown in Figure 8, chains of functions (tasks) create vThings that interface with the pub/sub system of VirIoT. Each task can autonomously initiate request/response communications to retrieve IoT device readings from the Root Data Domain, and to prepare vThing values to be served by the task. By means of the cache capability of ICN, the request/response communication might not, in some cases, propagate all the way to the Root Data Domains or to the real IoT devices, saving network resources and reducing latency.

The main idea is to "pass around" the NGSI-LD Entities that are part of vThing data values over request/response protocol of ICN. On the other hand, control messages are still transmitted by IP-based communication, since the tasks are implemented exploiting container technology, currently Docker, and the control is performed through the interface to Docker containers. Definitions of ICN messages that could be used for task control are not yet mature at the time of writing this delivarable, so that we briefly discuss preliminary ideas about information binding for exchanging NGSI-LD data payloads, but not for control messages.

IoT data retrieved from the Root Data Domain is converted to the NGSI-LD information model at the boundary of ICN, by the tasks connecting to the different data domains. The initiation of the boundary tasks is triggered by the downstream tasks in service function chains. The downstream tasks initiate autonomously using periodical

execution mechanism.

The request message (Interest packet) created by the initiating task has the format shown in Figure 9. The Content Name field (topmost field) may contain the NGSI-LD Entity Types that are requested for a certain vThing, thus identifying the elements that are going to compose it, which are going to to be retrieved by this request. The Function

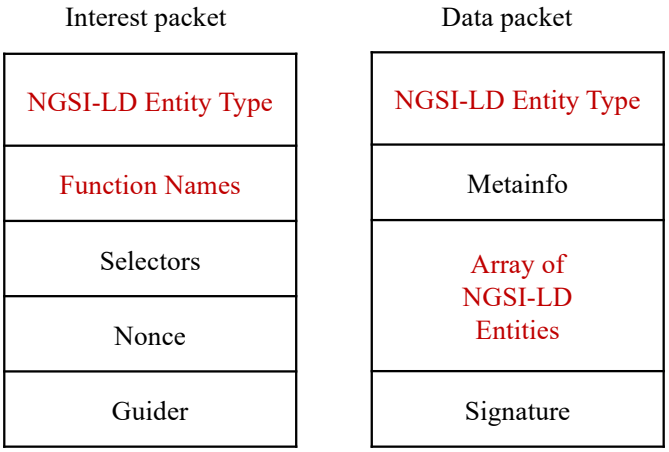

Figure 9: ICN Message Format for ThingVisor Factory.

Names field specifies the service function chain to be applied to the NGSI-LD Entities, matching the Entity Type, that embed the corresponding information derived from Root Data Domains.

The NGSI-LD Entities derived by applying tasks to the data of the Root Data Domain are placed in the Content field of response message (Data packet) as shown in Figure 9. What is contained in the Content field is typically going to be an array of NGSI-LD information.

Data packets may be cached at ICN routers as shown in Figure 8. If cached data exist, the Interest packet requesting the cached data is not further propagated to the original source. The Interest packet, instead, is matched by the cached data and the router holding the cached data responds with the corresponding cached Data packet. The lifetime of cached data need to be adjusted to the update interval of the original IoT device data so that tardy data is not delivered to the requesting tasks.

3 Cross-Domain Information Sharing

3.1 Introduction

The purpose of this section is to present the way information is transferred from a source of the Root Data Domain to one or several vSilos while minimizing information loss due to crossing through the whole VirIoT platform. As presented earlier in the document, the internal/neutral data model of the platform is NGSI-LD, which implies that any information within a Root Data Domain has to be converted to the NGSI-LD model by the corresponding ThingVisor. The reverse conversion, from the NGSI-LD model to the Root Data Domain model has to be handled similarly to prepare for handling of *actuators*, which will be discussed in the 2^{nd} release of the deliverable.

As an illustrative example, we can consider temperature information. In its simplest form, a temperature measurement would be expressed as a number (i.e. *22.8*), which, by itself, does not tell anything (is it a speed or a temperature? What is the unit that was used?). A more advanced scheme would associate a name to that measurement (i.e. *temperature*) and a unit (i.e. *°C*,) which would be understandable by a system using the same representation. Even more advanced approaches use ontologies as a way to capture and share a conceptualization of the information. In such approaches, information is traditionally encoded within a scheme that is aimed at limiting the information loss when sharing information.

3.2 SenML ⇔ NGSI-LD

3.2.1 SenML overview

Sensor Markup Language (SenML) is a low-energy-consuming language to retrieve measurements from M2M devices and a non-proprietary format. SenML provides simple measurements: the name, the units, and the value. SenML format allows information aggregation over a time interval providing eventually a GPS location. The accepted data formats are specified in SenML specification [3]. These are summarized below.

Simple form SenML representation. *The simplest form encoding measurement of 23.1 °C provided by sensor with id urn:dev:ow:10e2073a01080063*

```
[ {"n":"urn:dev:ow:10e2073a01080063","u":"Cel","v":23.1} ]
```

Multi-values form SenML representation. *The form encoding differents measurement (voltage of 120.1 V and current of 1.2 A provided by sensor with id urn:dev:ow:10e2073a01080063*

```
[{"bn":"urn:dev:ow:10e2073a01080063:","n":"voltage","u":"V","v":120.1},
  {"n":"current","u":"A","v":1.2}]
```

Periodic measurements SenML representation. *The form encoding periodic measurement (5 current measurement taken at 1s interval before timestamp 1.276020076001e+09 provided by sensor with id urn:dev:ow:10e2073a0108006*

```
[ {"bn":"urn:dev:ow:10e2073a0108006","bt":1.276020076001e+09, "bu":"A","
    bver":5, "n":"voltage","u":"V","v":120.1}, {"n":"current","t":-5,"v
    ":1.2}, {"n":"current","t":-4,"v":1.3}, {"n":"current","t":-3,"v
    ":1.4}, {"n":"current","t":-2,"v":1.5}, {"n":"current","t":-1,"v
    ":1.6}, {"n":"current","v":1.7} ]
```

3.2.2 NGSI-LD encoding

A NGSI-LD encoding of a SenML packet normalised to its simplest form would look like:

NGSI-LD sensor measurement. *Extract of NGSI-LD representation of a simple sensor measurement*

```
{
   "id":"urn:ngsi-ld:senML_Sensor:00YFZF",
   "type":"Temperature",
   "observedAt":"2010-06-08T18:01:16.001",
   "value":23.1,
   "unitCode":"CEL",
}
```

In the case of NGSI-LD, the measurement characteristics (timestamp, unit, value, etc.) are provided as *Properties*. A first issue we met is the identification of the measurement type. While a SenML measure expressed in *°C* is undoubtedly of type temperature, other units may not have such a simple mapping. As an example, a concentration expressed in *mg/l* does not say anything about the observed physical or chemical specie. In that case, the system has to be informed about the type of measurement provided by the sensor when it is registered into the platform. This mapping operation is realised at the ThingVisor level which also maps the senML sensor basename to the NGSI-LD sensor ID. The last mapping to be achieved is the unit code. The senML specification [3] provides a limited list of unit codes which needs to be mapped to the UN/EDIFACT common

codes [4] as recommended within the NGSI-LD specification. Table 5 provides examples of simple mapping between these two encoding schemes.

Unit name	senML Symbol	NGSI-LD recommendation
meter	m	MTR
kilogram	kg	KGM
gram	g	GRM
second	s	SEC
ampere	A	AMP
kelvin	K	KEL
radian	rad	C81

Table 5: Sample of senML to NGSI-LD unit code mapping

3.3 NGSIv2 ⇔ NGSI-LD

This section presents the mapping strategy between NGSI-LD and NGSIv2. The FIWARE data models had been studied to define the data model and the mapping process.

In NGSIv2, the entities only contain attributes. These attributes are understood as properties. They can be of different types (alphanumeric, date and time, etc.). Optionally each one may contain metadata.

In NGSI-LD entities contain properties and also relationships, optionally each one may include metadata. In NGSI-LD there aren't different types of properties. NGSI-LD offers a powerful means to encode the relationships between entities.

As NGSIv2 lacks structuring able differentiate between properties and relationships, FIWARE introduced the convention that the name of the attributes that represent a relationship should start with "ref". Also, if the type of the attribute is "Relationship" or "Reference", it will be considered as a relationship too.

Finally, as a result of the mapping process, from each NGSI-LD entity, a single entity will be obtained in NGSIv2 and vice versa.

Next, we can see an example of a parking site entity in NGSIv2 and its corresponding representation in NGSI-LD. The @context is omitted for brevity.

NGSI-v2 Parking site entity.

```
{ id: 'urn:ngsi-ld:parkingsite:Aparcamiento:101',
  type: 'parkingsite',
  timestamp: { value: '2019-04-29T12:30:00Z', type: 'DateTime', metadata:
      {} },
  name: { type: 'Text', value: 'Libertad', metadata: {} },
  numSpacePC: { type: 'Text', value: '51', metadata: {} },
  totSpacePCCapacity: { type: 'Text', value: '330', metadata: {} },
  maxHeight: { type: 'Number', value: 1.8, metadata: {} },
  maxLength: { type: 'Number', value: 5.1, metadata: {} },
  maxWidth: { type: 'Number', value: 2.3, metadata: {} },
  payMthd:
   { type: 'StructuredValue',
     value: [ 'Cash', 'PayPal' ],
     metadata: {} },
  payMthdCreditCard:
   { type: 'StructuredValue',
     value: [ 'AmericanExpress', 'Discover', 'MasterCard', 'VISA' ],
     metadata: {} },
  policyPC:
   { type: 'Relationship',
     value: 'urn:ngsi-ld:policy:Aparcamiento:101:PrivateCar',
     metadata: { entityType: { value: 'policy', type: 'Text' } } },
  isOpen: { type: 'boolean', value: true, metadata: {} },
  location:
   { type: 'geo:json',
     value: { type: 'Point', coordinates: [ '-1.1336517', '37.9894006' ]
        },
     metadata: {} } }
```

NGSI-LD Parking site entity.


```
{
    "id": "urn:ngsi-ld:parkingsite:Aparcamiento:101",
    "type": "parkingsite",
    "observedAt": {
        "type": "Property",
        "value": {"@type": "DateTime", "@value": "2019-04-29T12:30:00Z" }
           },
    "name": {
        "type": "Property",
        "value": "Libertad"
    },
    "numSpacePC": { "type": "Property", "value": "51" },
    "totSpacePCCapacity": { "type": "Property", "value": "330" },
    "maxHeight": { "type": "Property", "value": 1.80 },
    "maxLength": { "type": "Property", "value": 5.10 },
    "maxWidth": { "type": "Property", "value": 2.30 },
    "payMthd": {
        "type": "Property",
        "value": ["Cash","PayPal"]
    },
    "payMthdCreditCard": {
        "type": "Property",
        "value": ["AmericanExpress","Discover","MasterCard","VISA"]
    },
    "policyPC": {
        "type": "RelationShip",
        "object": "urn:ngsi-ld:policy:Aparcamiento:101:PrivateCar"
    },
    "isOpen": { "type": "Property", "value": true },
    "location": {
        "type": "GeoProperty",
        "value": { "type": "Point", "coordinates": [ "-1.1336517",
           "37.9894006" ] }
    }
}
```


3.4 oneM2M ⇔ NGSI-LD

The oneM2M[5] platform is a global standard initiative for Machine to Machine (M2M) Communications and the IoT, providing a common M2M Service Layer with common service functions to connect and interwork devices. The oneM2M architecture is composed of the following four functional entities: the application dedicated node (ADN); the

application service node (ASN); the middle node (MN); and the infrastructure node (IN). Each node contains a common services entity (CSE), an application entity (AE), or both. An AE provides application logic, such as remote power monitoring, for end-to-end M2M solutions. A CSE comprises a set of service functions called common services functions (CSFs) that can be used by applications and other CSEs. CSFs includes registration, security, application, service, data and device management, etc.

The NGSI-LD information model and API was presented in section 2.1.1.

Within this section, a Smart Camera use-case located in the city of Grasse will be presented and studied in both NGSI-LD and oneM2M formats, in order to briefly present the strategy for mapping from NGSI-LD to oneM2M (already introduced in previous deliverable D2.2) and then to study it in more detail for the reverse mapping, i.e. from oneM2M to NGSI-LD.

The context of this Grasse example use-case consists of a sensitive site monitored by a smart camera, which may be able to detect events of wild deposits and provides the location, the car plate number and the type of waste illegally deposited.

3.4.1 Mapping from NGSI-LD to oneM2M

The NGSI-LD data model is our starting point for mapping NGSI-LD to oneM2M. Figure 10 models this example in NGSI-LD.

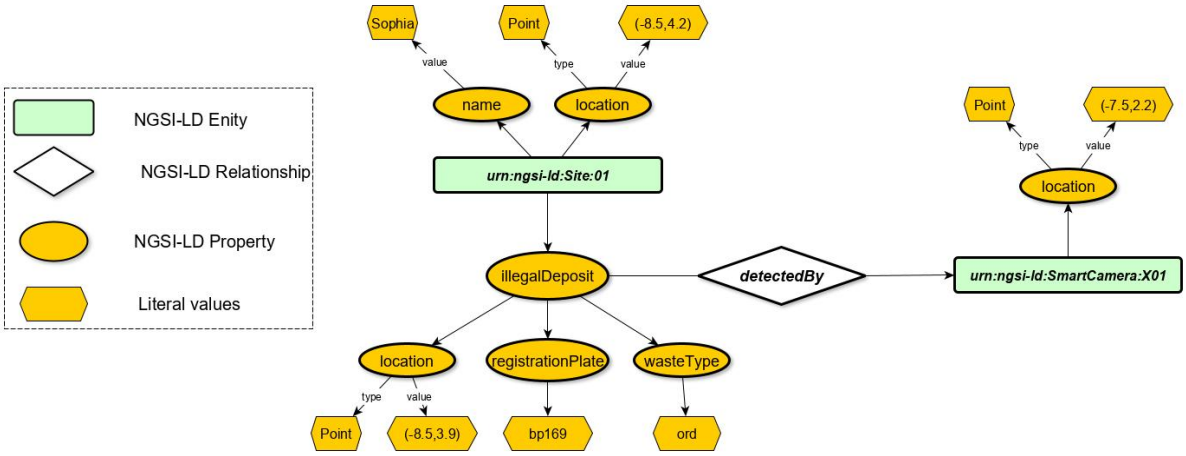

Figure 10: Grasse Use Case Example modeled in NGSI-LD

Below, the NGSI-LD entities of Site and Smart Camera. The @context is omitted for brevity.

NGSI-LD Site Entity.


```
{
    "id":"urn:ngsi-ld:Site:01",
    "type":"Site",
    "location":{
        "type":"GeoProperty",
        "value":{
            "type":"Point",
            "coordinates":[-8.5,4.2]
        }
    },
    "name":{
        "type":"Property",
        "value":"Sophia"
    },
    "illegalDeposit":{
        "type":"Property",
        "location":{
            "type":"GeoProperty",
            "value":{
                "type":"Point",
                "coordinates":[-8.5,3.9]
            }
        },
        "registrationPlate":{
            "type":"Property",
            "value":"bp169"
        },
        "wasteType":{
            "type":"Property",
            "value":"ORD"
        },
        "detectedBy":{
            "type":"Relationship",
            "object":"urn:ngsi-ld:SmartCamera:X01"
        }
    }
 }
```

NGSI-LD Smart Camera Entity.

```
{
    "id":"urn:ngsi-ld:SmartCamera:X01",
    "type":"SmartCamera",
    "location":{
        "type":"GeoProperty",
        "value":{
          "type":"Point",
          "coordinates":[-7.5,2.2]
        }
    }
 }
```

According to the previous work done in the deliverable D2.2, our adopted strategy for mapping NGSI-LD to oneM2M is summarized as follows: every NGSI-LD entity is mapped to a different Container within the same oneM2M Application Entity. For this purpose, NGSI-LD property values or Relationship object are mapped directly to oneM2M resources of type ContentInstance. The type (i.e Property and Relationship) is added as labels in the semantic descriptor for the container. Each NGSI-LD entity is mapped to a top-level container.

3.4.2 Mapping from oneM2M to NGSI-LD

The starting point for mapping oneM2M to NGSI-LD is the oneM2M resource tree, which is detailed in Figure 11.

The Semantic Descriptor, part of the oneM2M standard specification, enables attaching semantic description via RDF triples to the resource. These added triples are exploited as much as possible for mapping oneM2M to NGSI-LD. For this purpose, this section will focus on the impact of the semantic description on the mapping of oneM2M to NGSI-LD according to 3 main levels: (1) Mapping oneM2M to NGSI-LD without semantic description, (2) Mapping oneM2M to NGSI-LD allowing to freely choose the reference ontology, and (3) Mapping oneM2M to NGSI-LD with a specific reference ontology (NGSI-LD and SAREF).

Details about each of the possible alternatives are given in the following.

- Mapping oneM2M to NGSI-LD without the semantic description: In this case we consider a oneM2M resource tree with an empty (or not implemented) semantic descriptor. The @context is a crucial field in the NGSI-LD messages. In fact, the URIs of each property/relationship name have to be referenced in the @context file(s). To have a mapping of the oneM2M resource tree without at least having URIs of the CSEs, AEs, Containers and Sub-containers is not realistic in this case.

- Mapping oneM2M to NGSI-LD with free choice of the reference ontology: In this

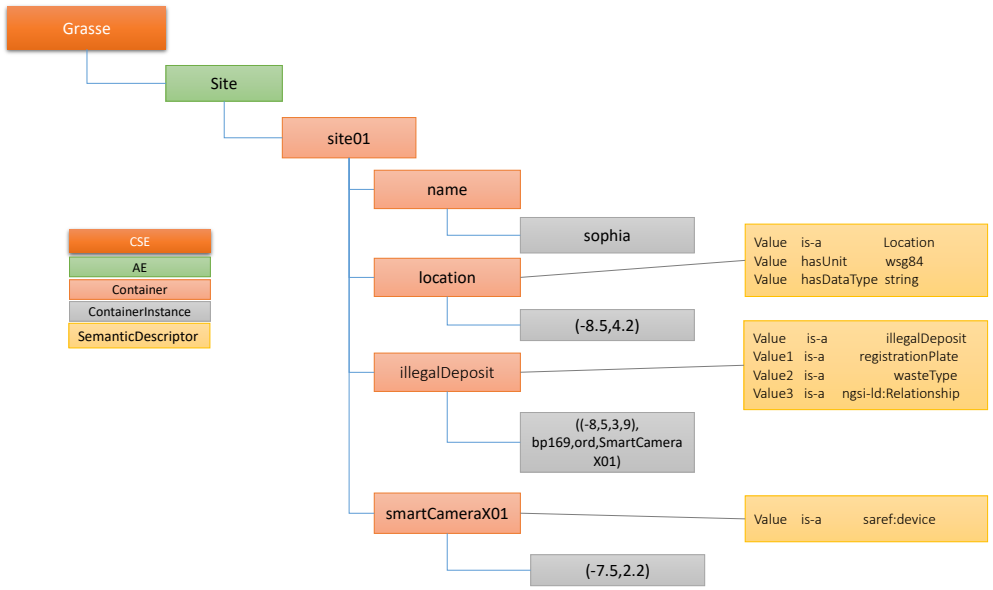

Figure 11: oneM2M resource tree of the Grasse Use Case Example

case we suppose that all CSEs, AEs, Containers and Sub-containers have an URI pointing to their semantic description. These URIs are directly added within the @context file, as a preliminary necessary step towards achieving a well-formed mapping. In Table 6 we summarize the further steps of our mapping strategy from oneM2M to NGSI-LD.

Table 6: First steps towards mapping oneM2M to NGSI-LD

oneM2M	NGSI-LD
top-level container	Entity
AE	Entity type
top-level container resource ID	Entity Id = "urn:ngsi-ld:"+AE+":"+resource ID
sub-container resourceName	Property name
content instance of the sub-container	Property value

Applying this strategy in our use case example, three main problems appear:

(1) Properties (location, registrationPlate, wasteType) of the Property "illegalDeposit" are all mapped to a unique value as follow:

```
"illegalDeposit":{
   "type":"Property",
   "value":"((-8,5,3,9), bp169,ord,SmartCameraX01)"
}
```

Properties of Property and Relationships of Property are not supported using this strategy.

(2) Relationships following this mapping strategy are not detectable, all sub-containers of the resource tree are seen as NGSI-LD properties and content instance of the sub-container as property values.

(3) The sub-container "smartCameraX01" has to be mapped to a new NGSI-LD entity not to a property.

- Mapping oneM2M to NGSI-LD with a specific reference ontology (NGSI-LD and SAREF): In order to produce a correct mapping strategy we have to refer to some known ontologies and critically to SAREF and NGSI-LD ontologies. Referring these ontologies in the semantic descriptor and adding new generic rules will allow us to a successful oneM2M to NGSI-LD mapping strategy.

  In the following, we try to resolve the previously detected issues by adapting the semantic description and referring NGSI-LD and SAREF ontologies.

Rules for detecting Properties of Property

```
If a sub-container is described as an RDF class or NGSI-LD Enity
   that have properties as domain, we can directly assume that this
    sub-container will be mapped as an NGSI-LD property, its
   semantic properties as NGSI-LD properties of the property and
   values of the container Instances as values of the properties.
```

The same strategy may be applied for detecting NGSI-LD Relationships of Property, especially when we apply the following rules for detecting NGSI-LD Relationships.

Rules for detecting Relationships

```
Each desired Relationship in the oneM2M resource tree has to be
   defined and specified as a NGSI-LD Relationship in the semantic
   descriptor by referencing the URI of the NGSI-LD relationship "
   https://uri.etsi.org/ngsi-ld/Relationship".
```

The Relationship concept is defined and introduced by the NGSI-LD standard, and detecting it in the oneM2M resource tree or even in the semantic description is hard task if the NGSI-LD URI is not referenced. When an NGSI-LD Relationship is detected its value will be added to the "object" field and a new Entity will be created. Properties of this newly created Entity are not known and have to be brought in. That's why a sub-container with the properties of the Smart camera was added.

Rules for detecting new Entities

```
Each saref:device is mapped to a new NGSI-LD Entity where its type
    is the label of the sub-container, the Id is the resource Id of
    the sub-container. Content instances of the sub-container are
    mapped as properties of the new Entity.
```

The SAREF ontology was mentioned in the previous section as a solution for detecting new entities in the oneM2M resource tree and thus mapping all SAREF properties of the semantic description to NGSI-LD properties. As mentioned before, Relationships have to be declared, in order for them to be detectable and thus mapped to NGSI-LD Relationship correctly.

4 Analysis of MQTT Clusters

An MQTT cluster is a distributed system that behaves, from the user point of view, as a single logical broker while multiple MQTT nodes handle the workload under the curtains. We plan to use a MQTT cluster for VirIoT internal information sharing (Figure 1) because the advantages are multiples, from the increase of reliability and availability of the services to the achievement of better performances. The effectiveness of the clustering technique can be observed to handle failover scenarios: if one of the brokers is not available, the remaining working nodes are still able to work properly avoiding a system-wide outage. This is achieved when the brokers work together and the system can route correctly the incoming traffic to the running brokers. It is now understood that the broker plays an essential role in MQTT communications. So critical in certain scenarios, that if not clusterized can lead the whole system to break down. Commonly, nodes run in distinct physical servers or are hosted in virtual machines, preferably connected over a dedicated network to isolate the traffic between them and the outside world.

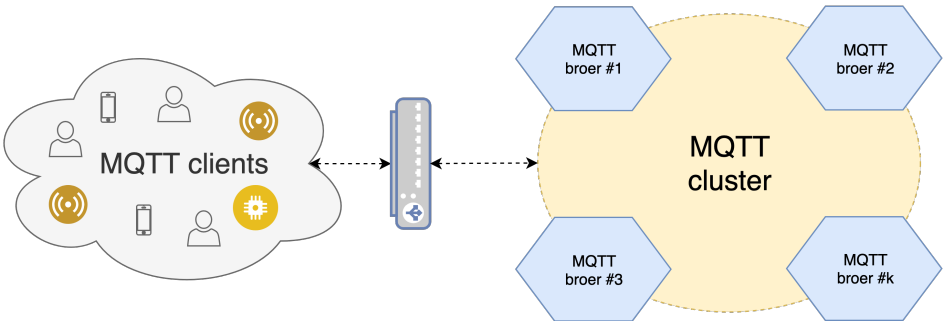

Figure 12: Cluster example

In general, an MQTT broker is a stateful application, meaning the nodes store information about the MQTT sessions necessary for the protocol to work properly, like connected clients, updated topic tables and also information about the cluster itself. A specific case is when a node stops functioning, the cluster must work it around and deal with a netsplit, which is the result of a failure of one or more network devices causing nodes no more reachable. Thus, the cluster must be aware of it and adjust to the situation accordingly. It will know that there is a node missing and it will keep looking for it, until the node comes back online and heals the partition or the other way around, it is forced to leave otherwise an eternal netsplit will arise.

The nodes become aware of the elements of the cluster through a discovery phase and identify each other using a unique name, which is used to join in the cluster and leave it later on. When a new subscription is directed to an MQTT broker, it is the job of the latter to inform all the other nodes it has the client subscription to the specified topics.

Whereas an MQTT client publishes a message, the node will search a match onto the topic table and forwards the message to all the nodes owning these clients.

So all the nodes must have a synchronised version of the topic table, needed to perform a lookup match every time a new message comes in. Indeed, this operation will perform traffic across the cluster. The lack of this functionality precludes the implementation to work in a cluster fashion way. The broker also holds the sessions of all persisted clients, including subscriptions and missed messages.

One popular open-source implementations of MQTT broker is *Eclipse Mosquitto* [6]. It is written in C and thanks to its small code footprint is ideal for the use on almost any device from low-power embedded devices to full servers, but it has some drawbacks. It does not support clustering and can't take advantage of multi-core CPUs as it leverages only one single thread. For these reasons, we didn't consider it for our studies, since the clustering ability was essential.

An interesting high-performance, distributed MQTT implementation, is *VerneMQ* [7], based on Erlang OTP, a language very popular in the message broker world because of its distributed capabilities for building highly scalable messaging systems. This enables VerneMQ to scale both horizontally and vertically. It fully supports clustering capabilities, supporting a high number of concurrent publishers and subscribers while maintaining low and predictable latency. Its fault tolerance capabilities must be noted, specifically VerneMQ can tolerate incredibly well network failures and provide detailed control over availability and consistency of the brokers. Moreover, the underlying distributed data storage features automatic conflict resolution and recovers automatically from netsplits.

Another widespread message broker is *RabbitMQ* [8], it is written in Erlang and has support for clustering. The problem with RabbitMQ is the MQTT support itself. Although it supports natively other publish/subscribe protocols, like AMQP (*Advanced Message Queuing Protocol*), the MQTT implementation is missing some important features such as QoS2, which can be crucial in some cases.

Lastly, *HiveMQ* [9], built in Java with scalability and enterprise-ready security in mind, offers similar characteristics of *VerneMQ*. Unfortunately, it was not considered for our tests because not open-source and a license is necessary to make it properly work.

4.1 The scaling issue we found

Some experiments carried out with different open-source MQTT cluster implementations, namely VerneMQ [7] and eMQTT [10], showed similar non-perfect scaling problems that intrigued us to investigate the matter. This result has further strengthened our confidence in believing the issue is not related to the implementation, rather associated with the more generic and widely used clustering scheme presented in Section 4.1.1.

Let us describe our experiment. We have considered different MQTT cluster configurations, ranging from a cluster with one broker, up to a cluster with 4 brokers. We have deployed the cluster across the nodes using Kubernetes [11].

The pub/sub scenario comprised 1000 topics, each one with a single publisher and a single subscriber. Publishers are equal to each other, generate messages with a pay-

N.of brokers	VerneMQ (msg/s)	eMQTT (msg/s)
1	4000	2000
2	5035	2600
3	7035	3600
4	9030	4400

Table 7: Publishing rate providing 2 *ms* of latency versus the number of brokers of the cluster

load of 200 bytes, with MQTT QoS equal to 0, and with a message inter-time which follows a Poisson distribution. The brokers and the load balancer run on different virtual machines with 2 CPUs each, hosted by Microsoft Azure cloud. The publishers and the subscribers' applications run on another virtual machine with 16 CPUs, so that the message throughput bottleneck are, inevitably, the brokers.

We measured the scale-out benefit concerning the maximum publishing rate the system can sustain while keeping the average message delay lower than 2 ms. Related measurements are reported in Table 7.

To understand the obtained results, we defined the *expected* publishing rate as the measured publishing rate in case of a single broker, multiplied by the number of brokers in the cluster. For instance, Table 7 shows that the publishing rate providing 2 *ms* of delay with a single VerneMQ broker is 4000 messages per second, then the expected rate in case of two VerneMQ brokers is 8000 messages per second, and so forth. Naively, performance is expected to scale up linearly with the available brokers.

Surprisingly, measurements unveil that the actual performance is rather far from the expected behaviour. For example, with VerneMQ, in case of 4 brokers, we got a publishing rate of 9030 messages per second versus an expected one of 16000. Fig. 13 shows the ratio between the measured and the expected publishing rates. When the cluster size increases, the measured performance dramatically falls to approximately half of the expected outcome. In this configuration, the MQTT cluster is wasting half of the computing resources.

The motivations of this result will be examined in more detail in the following section.

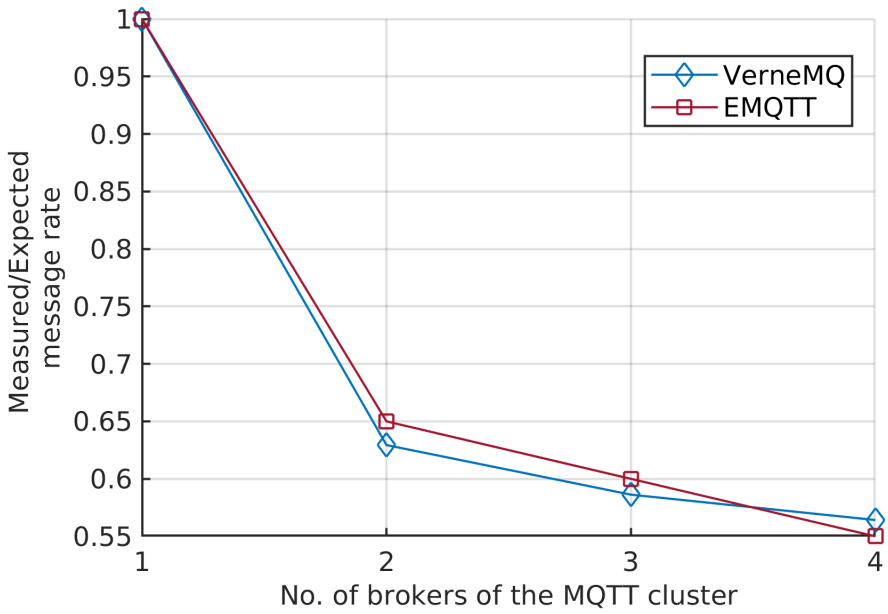

Figure 13: Measured/Expected message rate ratio for 2 *ms* latency, in case of 1000 topics, 1000 publishers and 1000 subscribers

4.1.1 Sub-linear performance scaling

To understand the motivations of the sub-linear scaling behavior of the cluster, we have firstly found out which are the configuration parameters having an impact on the latency introduced by a single broker. Thereafter, we have analyzed how these parameters behave in a cluster configuration.

In our testbed everything (broker, clients, etc.) runs in the same data-centre thus the network delays are negligible and the message latency is mainly due to the processing work carried out by the broker during the forwarding of the messages from the publishers to the subscribers. Such processing can be roughly split into two procedures: *matching* and *dispatching*.

The *matching* procedure is used to single out which are the subscribers interested to an incoming message, usually exploiting fast matching algorithms (e.g. subscription tries). Subsequently, the *dispatching* procedure is used to send the message to these subscribers, while handling related QoS. The execution of these procedures load the CPU as follows:

- the processing load due to the execution of the *matching procedure* depends on the number of messages (publication) per second received by the broker, hereafter named *input traffic*;
- the load due to the *dispatching procedure* is instead related to the number of message per second sent out by the broker, hereafter named *output traffic*.

We measured the impact on the *latency* of these two traffics by separating their effects. Figure 14 shows the latency versus the input traffic while the output traffic is kept constant at 2000 msg/s [3].

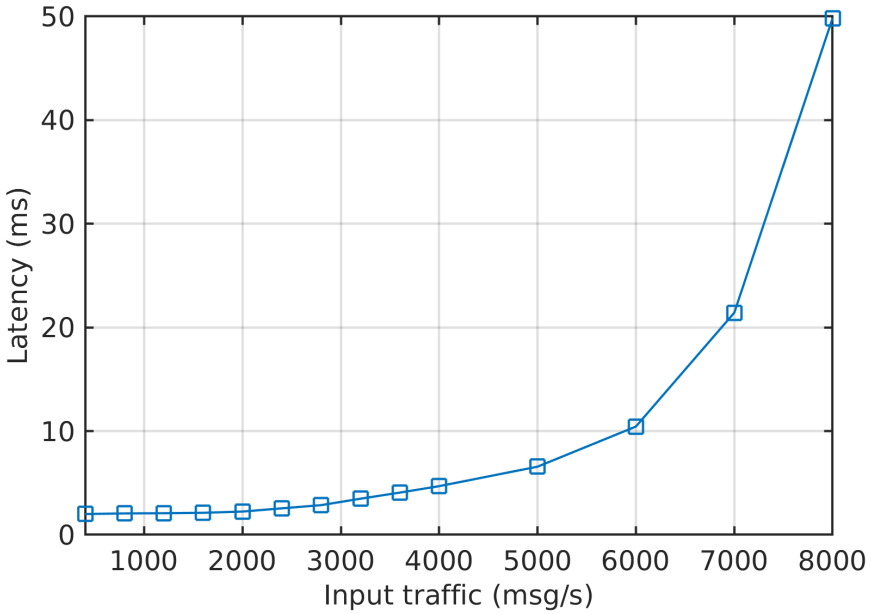

Figure 14: Message latency versus input traffic in case of VerneMQ single broker

Vice versa, Figure 15 shows the latency versus the output traffic while the input traffic is kept constant at 2000 msg/s [4]. As we can see the increase of the input and/or of the output traffic leads to an increase in the latency introduced by the broker.

[3]To reproduce this configuration we varied the number of topics from 100 to 2000. Each topic is used by a different publisher, sending a message with an average rate of 4 msg/s. The number of subscribers is 2000 and each of them is interested in a single topic which is chosen using a round-robin strategy

[4]To reproduce this configuration we used 500 topics. Each topic is used by a different publisher, sending a message with an average rate of 4 msg/s. The number of subscribers is varied from 500 to 4500 and each of them is interested in a single topic which is chosen using a round-robin strategy

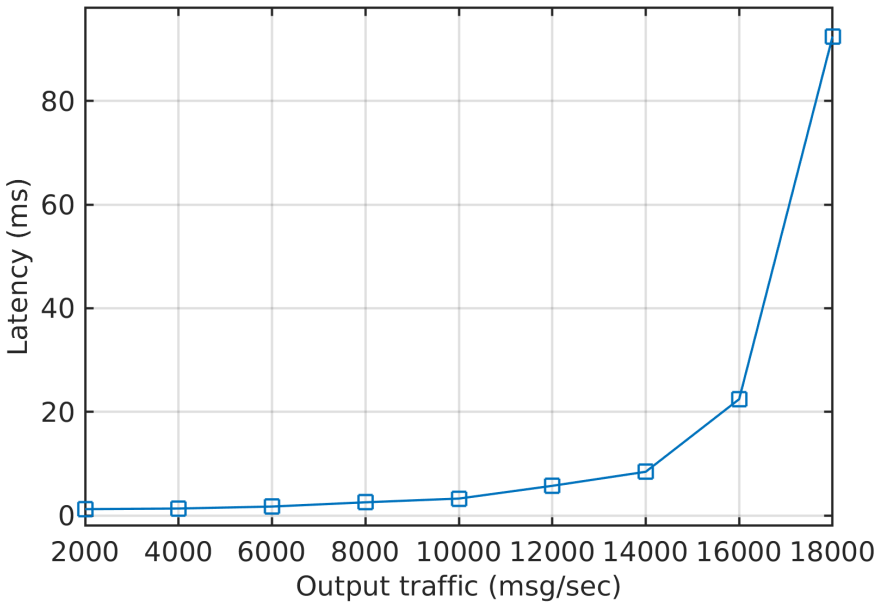

Figure 15: Message latency versus output traffic in case of VerneMQ single broker

In case of a single broker, the input traffic is simply composed by the stream of messages coming from the publishers connected to the broker, while the output traffic is formed by the stream of messages sent to the subscribers connected to the broker. In a cluster configuration though, in addition to these input/output *external* streams, we have an *internal*, node-to-node, input and output traffic that we argued to be the motivation of the not-perfect scaling issue since it increases the rate of matching and dispatching procedures thus the CPU load.

As shown in Figure 16, a generic broker k manages a subset of publishers and subscribers, and deals with four types of traffic:

- External input traffic (Aei_k) is the stream of messages generated by the connected publishers.
- External output traffic (Aeo_k) is the stream of messages sent to the connected subscribers
- Internal output traffic (Aio_k) is the part of the messages of the external input traffic forwarded to other brokers of the cluster since they have interested subscribers.
- Internal input traffic (Aii_k) is the stream of messages received from other brokers of the cluster because the broker k has interested subscribers.

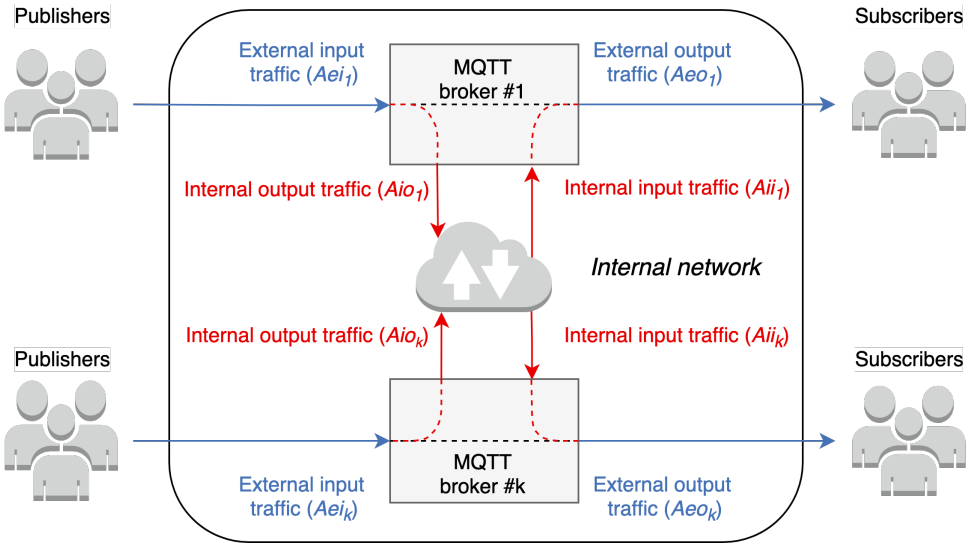

Figure 16: Internal and external traffic of an MQTT cluster

As shown by Figure 13, the increase of the number of brokers of the cluster leads to a proportional decrease of input/output external traffic, and thus of the CPU load, since the publisher/subscribers connections are uniformly distributed over a greater number of brokers.

On the other hand, Figure 14 shows that when the internal traffic shows up, the aforementioned CPU reduction partially vanishes.

4.2 Performance analysis

The goal of this chapter is to evaluate the impact of system and traffic parameters on the internal traffic obtaining important insights for the cluster design. We considered two MQTT use-cases namely Social Network and IoT.

The social network use-case is inspired by pub/sub social network applications like Twitter where a subscriber subscribes to different topics [12]. Accordingly, we assume that every subscriber is interested in several topics equal to $Nsxs$ (number of subscriptions per subscribers) and that the topic popularity follows a Zipf distribution with shape factor α.

The IoT use-case is inspired by a typical IoT environment in which topic names follows a hierarchy that is somehow related to the physical environment, e.g. *buildingId/floorId/roomeId/sensorType/sensorId*. Subscribers can be interested in the publications related to a specific topic or in the publications related to a cluster of topics sharing the same name-prefix, e.g. *buildingId/floorId/roomeId*.

We simply modelled this behaviour by setting up a tree of possible subscriptions (see Fig. 17) whose depth is d, every node has the same fan-out equal to f. The subscription selecting process provides that a subscriber first chooses a tree level according to a given

level probability vector $Pl = Pl_0..Pl_d$, where Pl_i is the probability of selecting the ith level. Then, the subscriber randomly chooses one out of the nodes of the level.

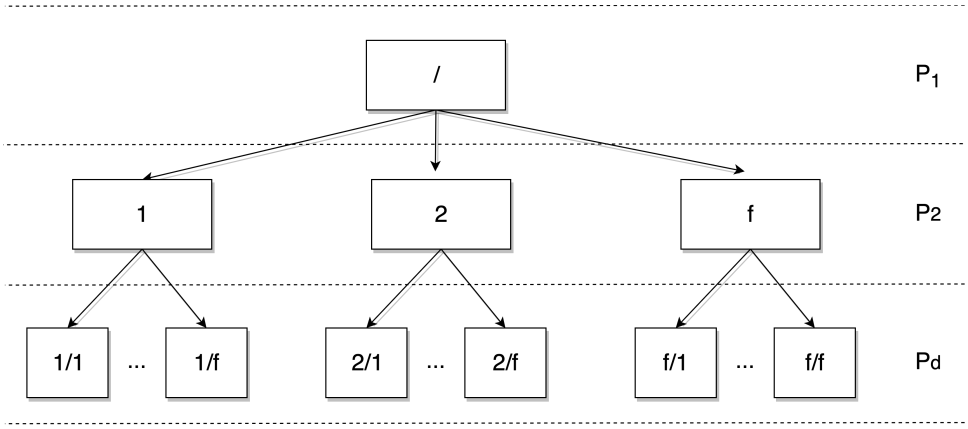

Figure 17: Subscription tree in an IoT scenario

We have developed a simulation tool using the MATLAB [13] environment to replicate the exact behaviour of the considered MQTT cluster. We assumed a publisher per topic which publish messages with a Poisson interarrival. Through the simulation analysis, we were able to study the behaviour of the traffic flows. In particular, we are focusing our attention on both the input and the output overhead of the traffic resulting from different cluster simulation scenarios, concerning the response a single node cluster would have.

We defined input and output overheads as follows:

$$overhead_{out} = \frac{A_{eo} + A_{io}}{A_{eo}} = 1 + \frac{A_{io}}{A_{eo}} \quad (1)$$

$$overhead_{int} = \frac{A_{ei} + A_{ii}}{A_{ei}} = 1 + \frac{A_{ii}}{A_{ei}} \quad (2)$$

where A_{ei} is the total external input traffic (the amount of publications per second received), A_{ei} is the external output traffic (the amount of publications per second sent to subscribers), and $A_{ii} = A_{io}$ is the total internal input (and output) traffic exchanged inside the cluster to transfer publications from the receiving node to the other nodes of the cluster having interested subscribers.

The input overhead is a measure of the additional amount of lookup per second every node of the cluster needs to do with respect to the case of a linear scaling. For instance if the input overhead is equal to 9 and we have 14 nodes, this means that in a perfect linear scaling the amount of lookup operations should be equal to $A_{ei}/16$, but due to inefficiency of the cluster each node actually carries out $9(A_{ei}/16)$ lookup (topic routing) operations per second, thus vanishing a lot the /16 load reduction expected by the horizontal scaling

of the cluster from 1 to 16 nodes. A same reasoning can be repeated for the output overhead with respect to the number of forwarding per second.

First of all, we wanted to simulate a typical use-case for the pub/sub pattern, that is why we are going to consider a scenario such as the one of a social network environment like Twitter where a subscriber subscribes to different topics [12], Spotify [14] or RSS feed [15].

To model our analysis we have chosen to consider some common parameters appropriate for the scenarios, constraining the model with the following degrees of freedom:

- α, the Zipf parameter;
- N_{sxs}, the number of topic subscriptions per subscriber;
- N_{sub}, the number of total subscribers;
- N_{top}, the set of topics available that correspond to the number of publishers;
- M, the number of cluster nodes available.

Through the following graphs, we are going to study the variation of some parameters, keeping the other fixed. The results were particularly valuable for our studies and the interpretation of them allowed us to have a better view of the big picture.

It has been shown [16] that it is reasonable to assume that the popularity of the topics subscribed by a subscriber follows a Zipf distribution similar to Web objects [17]. It follows that we have considered a value of 1.13 for the Zipf parameter α, to describe given distribution [16]. Other studies assumed a mismatched amount of publishers and subscribers, like [18] that to evaluate scalability, assumed one publisher changing the number of subscribers, while others authors, like [19], assumed the opposite consideration. We have chosen a configuration more suited for the social network scenario assumed before, with several publishers at least equal to the number of subscribers.

The behaviour of the previously mentioned Equations 1 and 2 define our model of study, that is why was helpful for us to plot both of the overheads, as well as the traffic parameters that compose it. One of the curves, the violet one, represents an algorithm that will be discussed in the last section, the greedy one. Furthermore, it can be seen from every figure that the model overhead, accurately represents the simulation undertaken, as it lines up almost perfectly with the simulated curves.

4.2.1 Study of the Cluster's Nodes Variation

First off, we wanted to see how the system would respond to a scale-out of the cluster's nodes in a steady scenario, where the number of clients does not change. In particular, we have tried the setting where the only parameter to change was the number of nodes to check if scaling would have a beneficial role in our considerations. It can be noted in Figure 19, that the two straight lines are the evidence of the initial constraints we have put: a constant input and a constant output of the entire cluster, not only on the

number of subscribers but even the number of the topics they are subscribed to, thus the publishers[5]. As might be expected, an increase of the cluster's size corresponds to an inevitable increase of the internal input overhead, defined by Equation 1. Its slope is ruled by the ratio between the green and the orange line (Figure 19) and corresponds to the orange input overhead in Figure 18. When the number of nodes becomes higher, the probability that a client publishes on a broker with the same topic's subscription becomes lower, thus the need of an additionally hop to send the message to the proper receiver, that increases the internal traffic.

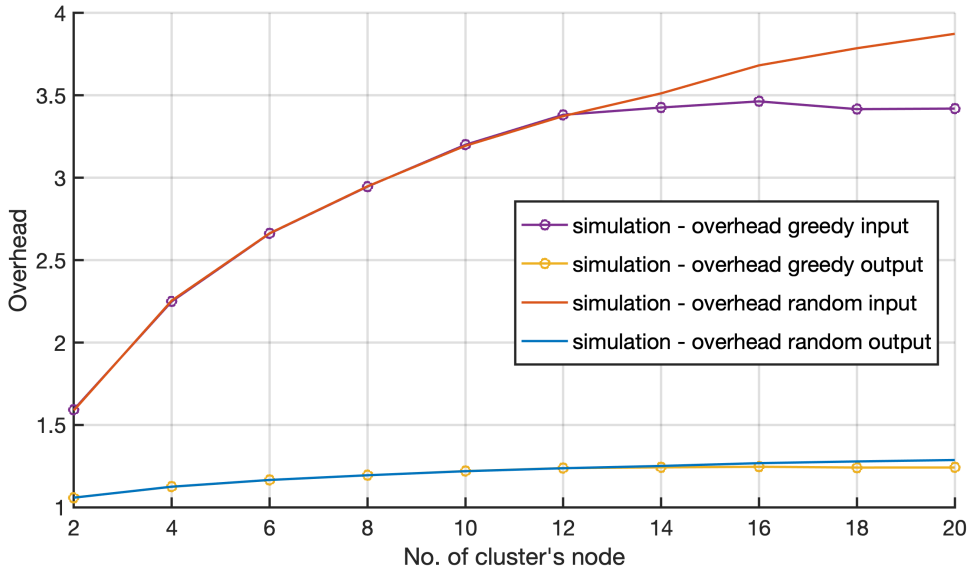

Figure 18: Overhead vs Number of nodes in the cluster

[5] $\alpha = 1.13, N_{sxs} = 10, N_{sub} = N_{top} = 5000$

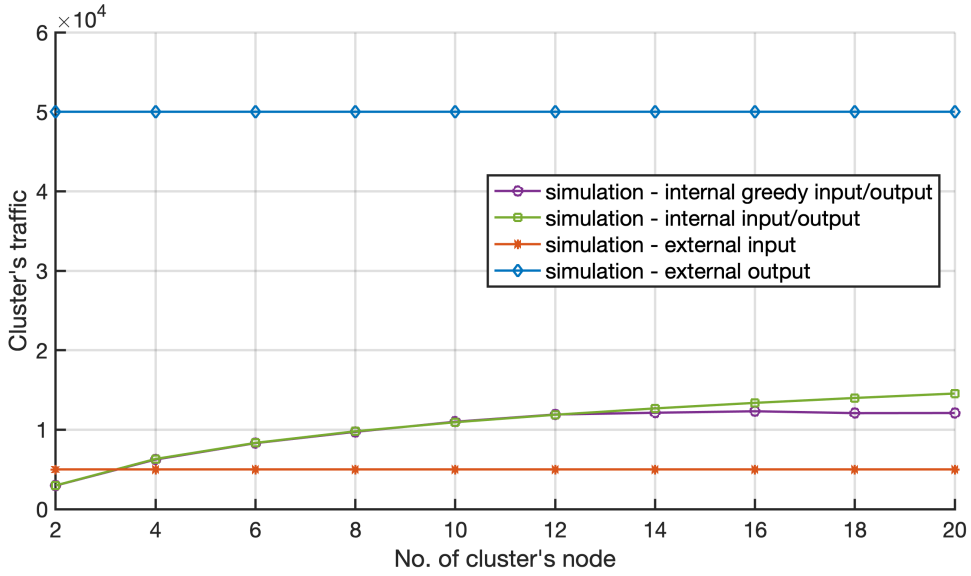

Figure 19: Cluster's traffic vs Number of nodes in the cluster

4.2.2 Study of the Cluster's Subscribers Variation

An example of this scenario could be the happening of an event, for instance, a national football match, where the interest of the match is higher than usual with a consequently higher number of subscribers of the topic, which is the match itself. To simulate this scenario as a sports matching application, we have considered the growing number of subscribers [6] while all the other properties of the cluster remain fixed, including the number of nodes. Clearly, this setting will affect the output traffic, as the amount of messages towards the external side grows linearly with the clients. As mentioned at the beginning of this section, the default value for the number of the subscriptions a subscriber is allowed to be interested in (N_{sxs}) is set as a default value of 10. This explains why the external output is higher, of an order of magnitude, concerning the other traffics plotted in Figure 21. This considerable difference will dramatically bring the output overhead to the unit, as it is the denominator in the Formula 2 it will cancel out the contribution of the internal traffic.

Similar behaviour can be obtained if we consider the variation of the previously mentioned N_{sxs}[7]. In other words, as the Figures 20 and 21 strongly pinpoint, the number of topics that draw the attention of the subscribers increases the number of flows addressed in the direction of the external subscribers, thus the dispatched traffic towards the outside, increases.

[6] $\alpha = 1.13, N_{sxs} = 10, M = 4, N_{top} = 5000$
[7] $\alpha = 1.13, M = 4, N_{sub} = N_{top} = 5000$

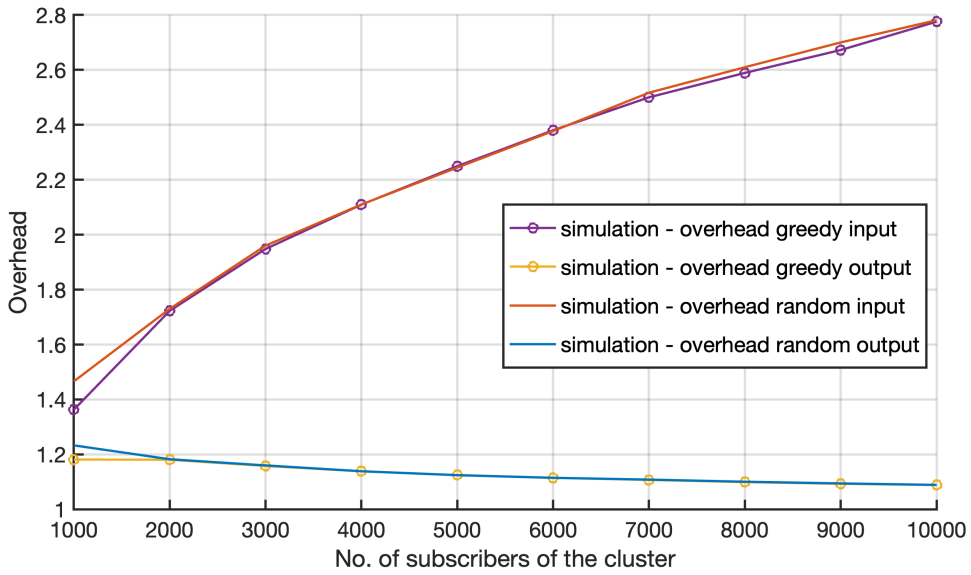

Figure 20: Overhead vs Number of subscribers

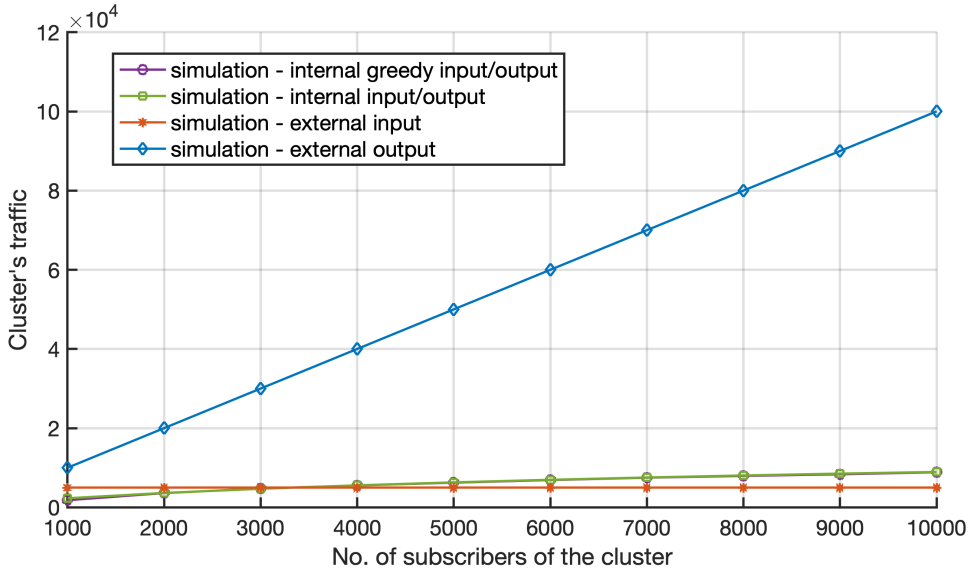

Figure 21: Cluster's traffic vs Number of nodes in the cluster

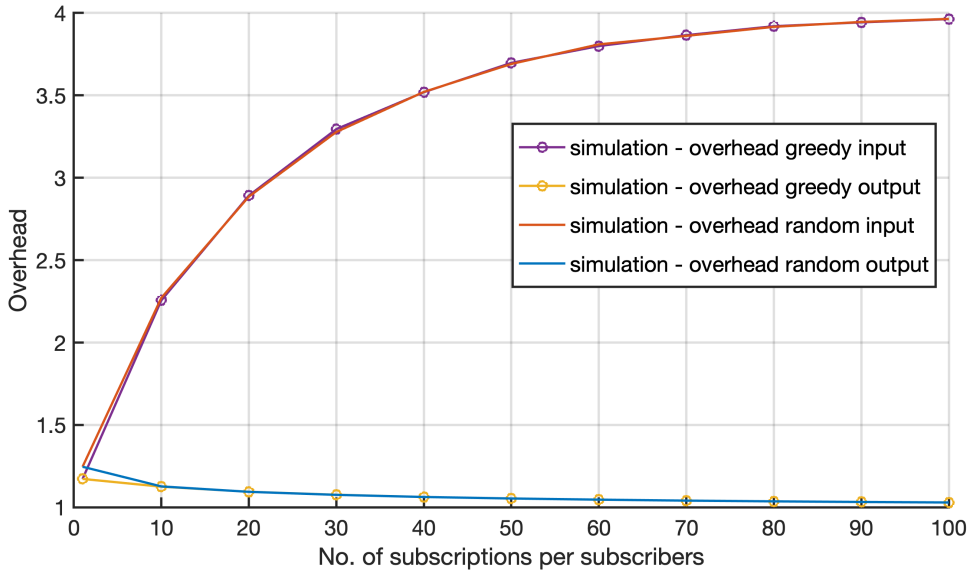

Figure 22: Overhead vs Number of subscriptions per subscriber

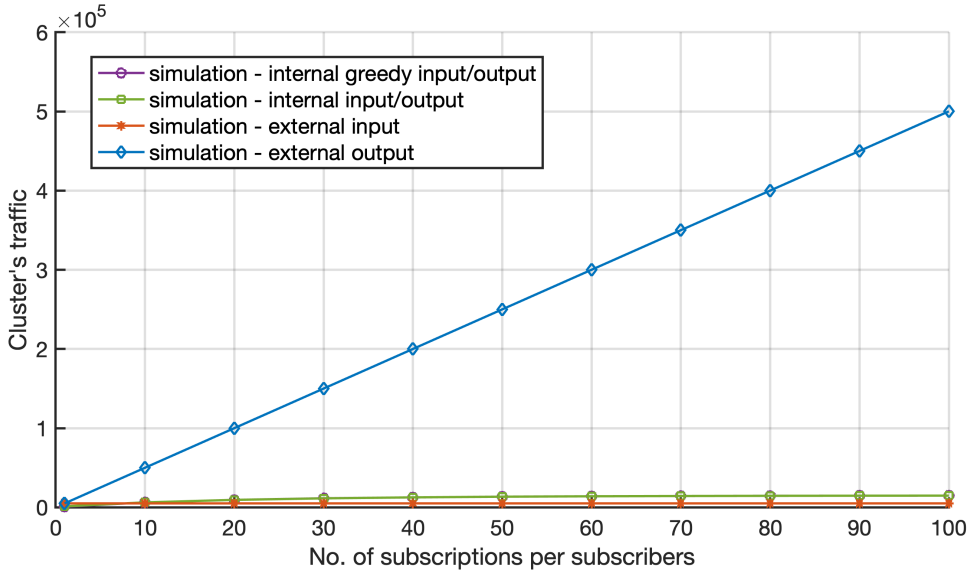

Figure 23: Cluster's traffic vs Number of subscriptions per subscriber

4.2.3 Study of the Cluster's Publishers Variation

One of the most significant interpretations of the results came from the scenario where the number of publishers grows. We have already covered the considerable impact the input traffic has on the latency when we presented our initial real measurements. Notably, we have discussed the type of processing that has an intense load on the CPU, stating the

heavy role of the matching procedure. With the help of the measurements we have seen the impact on the latency given by the input traffic, that is way heavier than the load given by the dispatching procedure.

Since the only traffic towards the external of the cluster remains constant, as the number of subscribers and their subscriptions, it is illustrated in Figure 25 as a straight line. On the other hand, the input traffic grows with the evolving of the simulation, as expected[8].

The probability a publisher needs an extra hop to reach its subscriber grows as the internal flows increase but, the increment is not radical as the one seen with the growing number of subscribers to feed. This is confirmed by Figure 21, since the input flows increases but slower than the external input ones. The increment is emphasised when the difference between the input and output flows becomes significant. This is the case of the end behaviour of Figure 25, where the global input traffic is more than the double of the internal one. So high that the cluster acts closer to a single node cluster rather a 4-node cluster, as the internal flows are almost completely neglected by the amount of the input coming from the external side.

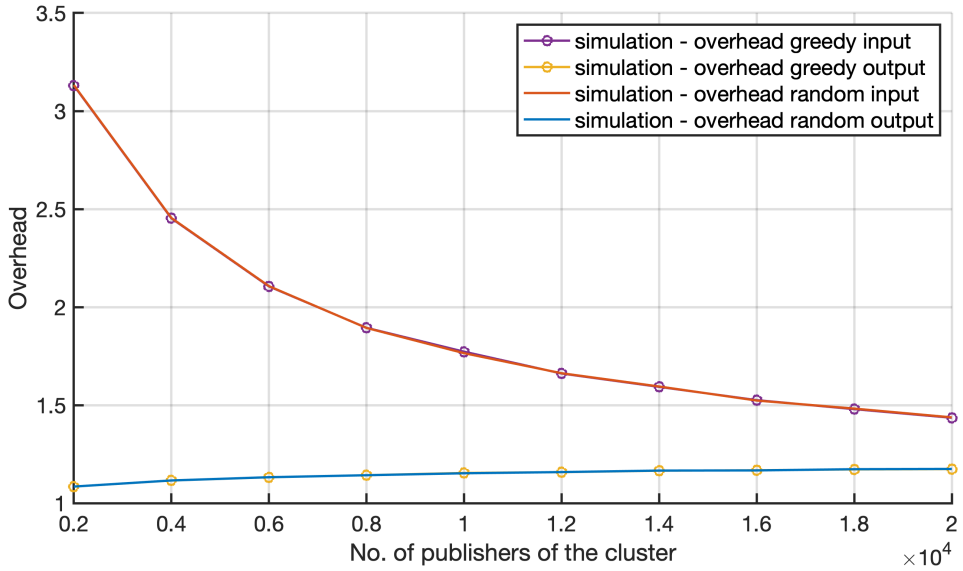

Figure 24: Overhead vs Number of Publishers

[8] $\alpha = 1.13, N_{sxs} = 10, M = 4, N_{sub} = 5000$

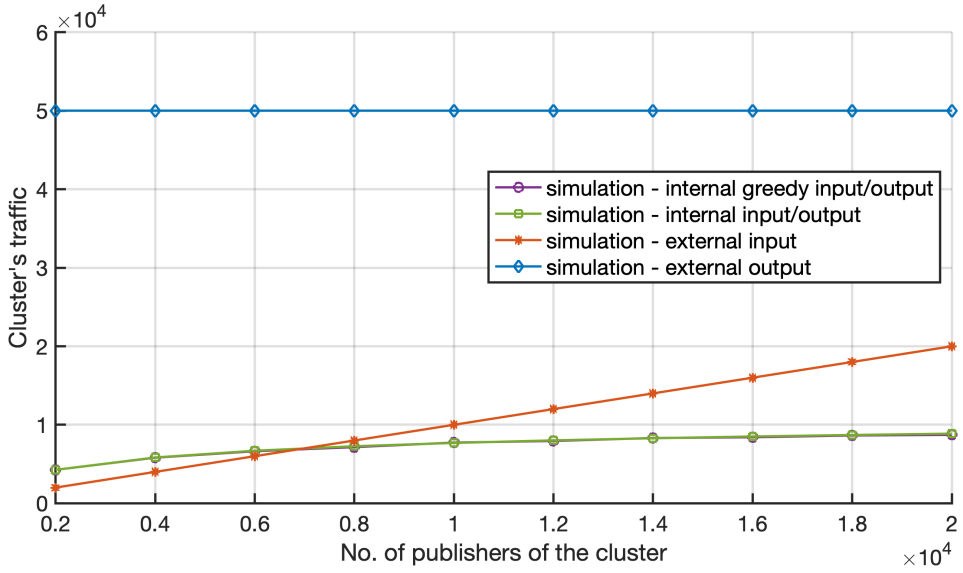

Figure 25: Cluster's traffic vs Number of Publishers

4.2.4 Study of the Zipf Parameter Variation

Finally, we have studied how the overall topic distribution influences the behaviour of the cluster. We issued subscriptions from a set of topics generated with different values of the Zipf α parameter[9]. From a uniform distribution of the topic's popularity, using a lower α parameter, to a distribution where fewer topics have a higher relevance, having the Zipf parameter *alpha* close to 3. The aforementioned behaviour is well illustrated by Figure 26, where the cluster tends to behave like a single cluster node when the choice of the topic, aka the popularity of a smaller set of topics, is highly increased. This is translated to a significant decrease in the internal traffic flows, as the interest for common topics is weakened, concurrently the probability a publisher has to choose a node with the corresponding subscribers has risen.

[9] $N_{sxs} = 10, M = 4, N_{sub} = N_{top} = 5000$

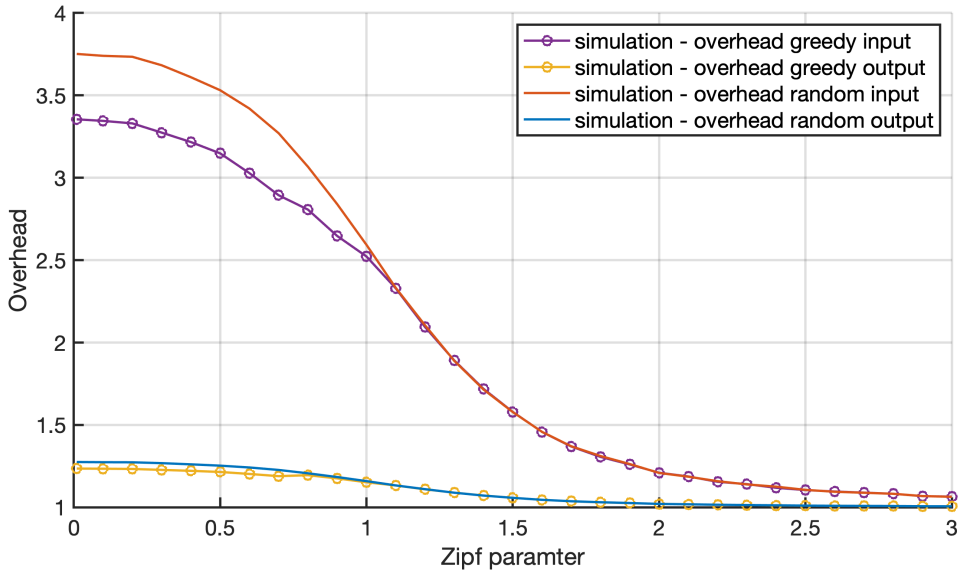

Figure 26: Overhead vs the Zipf parameter

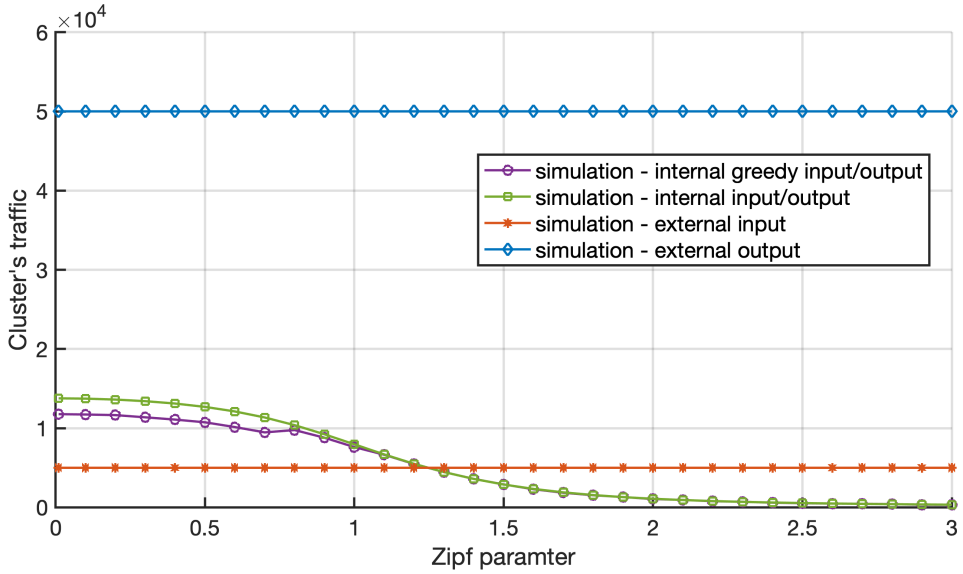

Figure 27: Cluster's traffic vs the Zipf parameter

4.2.5 Study of IoT scenario

In this section, we are going to explain how we have modelled the simulations for the IoT scenario mentioned in Section 4.2. First off, we have changed the way a subscriber can subscribe to a topic in an IoT fashion. Namely, we used a topic hierarchy to reproduce

an IoT environment related to a physical setting, e.g. *buildingId/floorId/roomeId/sensorType/sensorId*. In this way, a subscriber not only can subscribe to a single topic but can be interested in a higher level topic that embeds a given number of subtopics. Indeed, the amount of subscriptions increases, with a given probability, as the user chooses more topics than in the previously considered scenario.

We have undertaken the simulation in a steady scenario, where the number of clients and topics does not change, to see the response of the system to a scale-out of the cluster's nodes. From Figures 28 we can confirm the overhead has similar behaviour to the one in the social network scenario. The noticeable differences can be seen in the internal traffic flows, in Figure 29. Its behaviour is one of the green line, that grows with respect to the one in Figure 19 because of the properties of the set of topics. This means that a subscriber can be interested in a topic with a considerable number of subtopics. This follows an inevitable increase of the internal traffic since there could be a publication on the higher-level topic that includes all the lower-level ones, or a publication on these. As a result, the probability of internal flows increases considerably.

It is remarkable to note the evolution of the traffic flows as we increment the fanout of the topics. Without a doubt, this simulation would bring a relevant rise of the streams of the messages between the brokers. Indeed, Figure 31 highlights this behaviour, but also the growing importance of the input traffic that brings the cluster to behave like a single node when the connections are significantly high. It can be noted that the increasing pace of both the internal and the external output traffic, grows with the number of fanout almost the same, keeping the overall output traffic at bay.

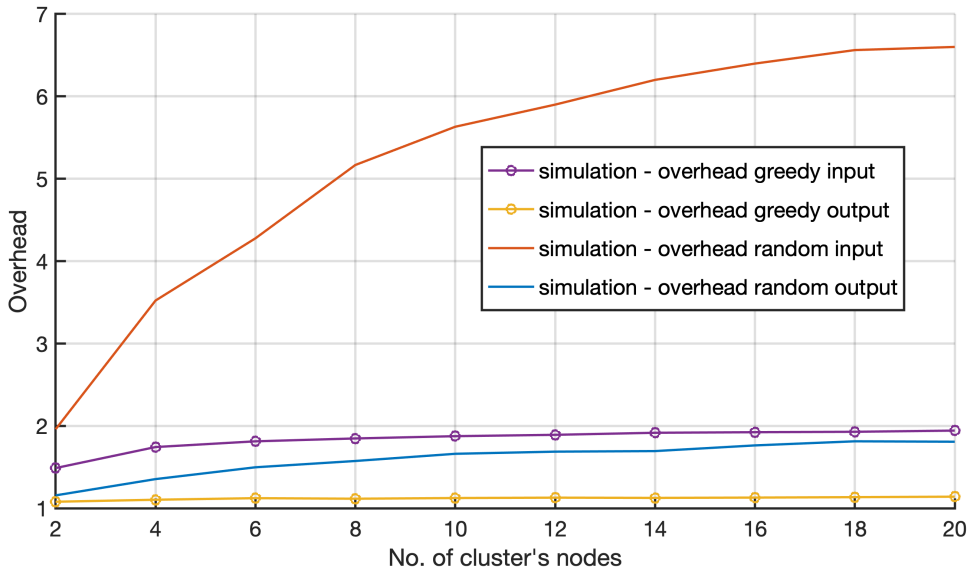

Figure 28: Overhead vs The number of nodes in the cluster

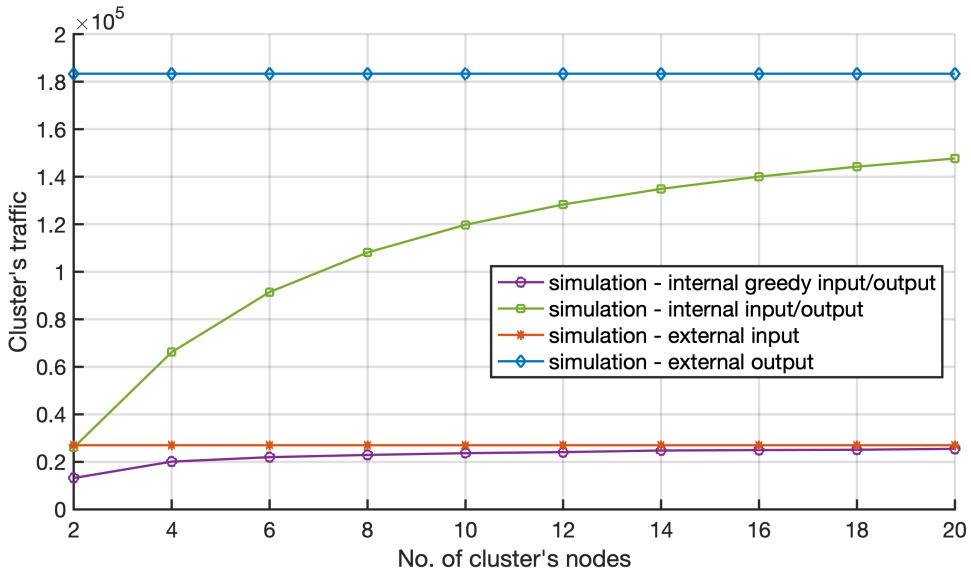

Figure 29: Cluster's traffic vs The number of nodes in the cluster

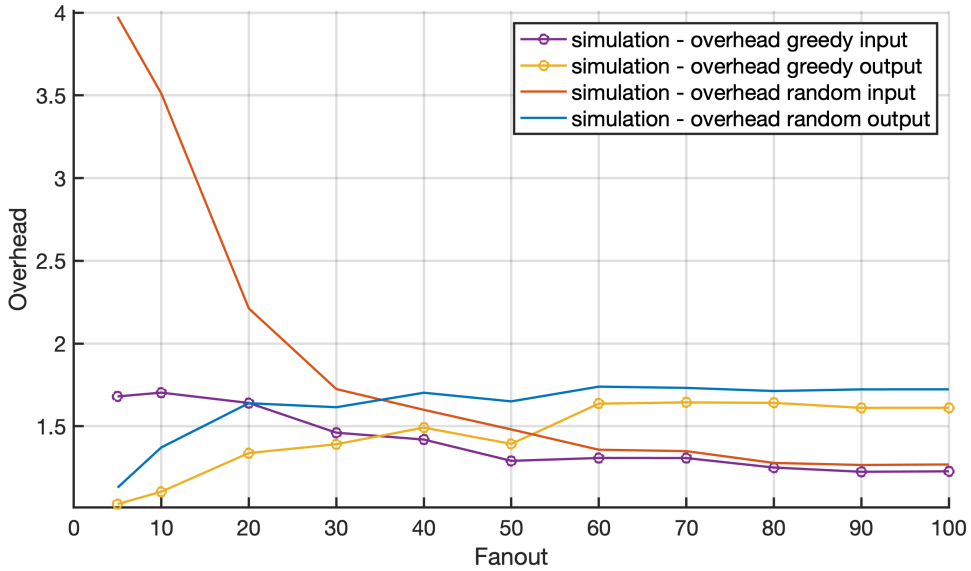

Figure 30: Overhead vs the fanout

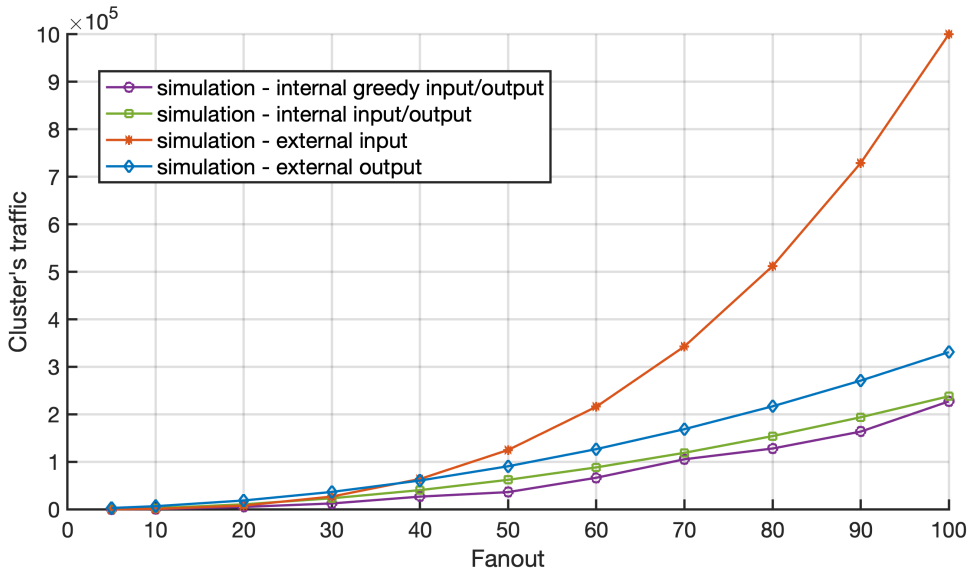

Figure 31: Cluster's traffic vs the fanout

4.2.6 Greedy Algorithm

At the beginning of the section, we have pointed out we have used two kinds of algorithms for the distribution of the subscribers among all the cluster's node. Up until now, we have discussed the outcome of the random algorithm, in this section it will be explained the second one, the greedy algorithm. It can be seen as a preliminary solution to reduce the internal traffic and its effect to the performance of the cluster will be taken into account.

The goal of the greedy algorithm is to try to minimize the internal traffic flows of the cluster as much as possible in order to reduce them close to the one node scenario taken as a reference. As well as being able to properly scale the performance together with the size of the cluster.

In order to achieve a greedy configuration of the subscribers among the cluster's node, the simulation tries, in a way, to clusterize the interests of the alike topics. To be more clear, the algorithm associates to the same node, starting from the first one, the subscribers with the greater number of similar topic's subscriptions in a water filling fashion way. The algorithm is going to group, as much as possible, similar subscribers in the fewest number of nodes, instead of spreading them uniformly at random.

Undoubtedly, the efficiency of this algorithm, which is its ability to correctly clusterize similar subscribers, strongly depends on the capacity of the node, as well as on the way the subscribers picks up a topic and its popularity.

All the figures confirm the inability to clusterize when the topics are sparse. As a matter of fact, the behaviour of the internal traffic of the cluster, when using the greedy algorithm, is almost the same as when the random one is used. We can see from the figures than only in Figures 18 and 19 we can catch a difference from the $12 - th$ node, as the algorithm starts to clusterize correctly. We can emphasise this behaviour choosing

a smaller number of subscribers as well as publishers and realize Figures 32 and 33. It is striking the differences with respect to the internal traffic shown in green, which is almost the quadruple for the end behaviour of the graph.

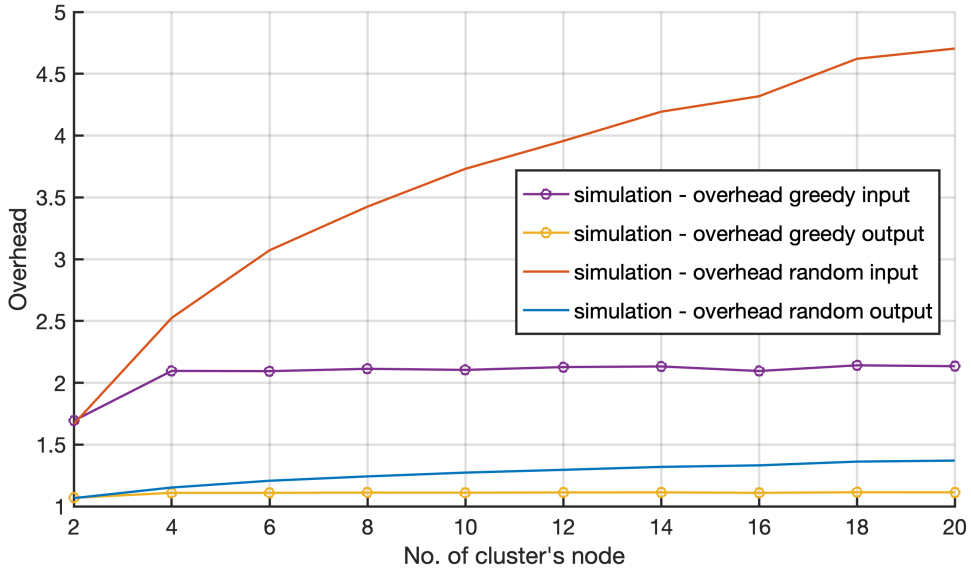

Figure 32: Overhead vs Number of nodes in the cluster, 1000 subscribers

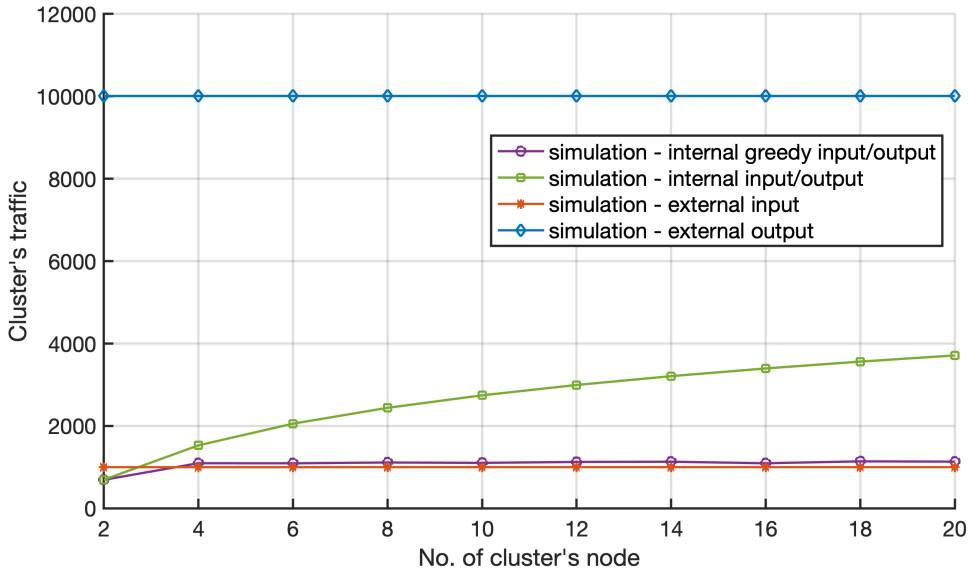

Figure 33: Cluster's traffic vs Number of nodes in the cluster, 1000 subscribers

Indeed, this is not a solution valid for all the configurations to guarantee a linear scaling of the performance with the cluster size but is a solution that works well for a set

of subscribers that have a certain similarity in choosing the topics.

This behaviour is significantly confirmed in the IoT scenario, where, for instance, the probability of subscribing to a subtopic is definitely higher than subscribing to its root tree topic, as Figures 30 and 31 notably show. The figures show how the algorithm better performs in this kind of scenario rather than the previously considered with a sparse set of topics.

4.3 Conclusions and impact on the VirIoT architecture

The central problem addressed in this section is the performance behaviour of MQTT clusters (e.g. to be used in the VirIoT architecture) when scaling out the number of cluster nodes. To start understanding the issue, we undertook some measurements considering different cluster configurations, ranging from a cluster with one broker, up to a cluster with 4 brokers using different MQTT implementations. The results suggested a behaviour not dependent on the implementation and showed a sub-linear evolution of the cluster's performance with the scaling of the brokers. Notably, to maintain the same performance, in terms of latency, the cluster tends to satisfy only half of the expected throughout. The above-mentioned outcome suggested us digging deeper into the matter, more specifically to understand better the behaviour of the internal traffic flows of the cluster.

To study a significant amount of examples, we have developed a MATLAB simulator. In this way we were able to study better the issues in two different pub/sub scenarios: the social network and the IoT, simulating them with some real use-case examples.

From the resulting simulations, we have highlighted the behaviour of the cluster's traffic as well as confronting it with the one of the single node schema. As a result, we have proposed an initial naive greedy algorithm which, however, provided limited performance enhancements, especially when the number of topics per subscriber increase, as it may occurs for a vSilo with many vThings.

Consequently, we propose a more radical topic isolation approach for the future release of the VirIoT architecture (see next D2.3, D3.2, D4.2) , where topics are strictly bounded to different brokers of the cluster and architectural entities must connect to more than one broker. For instance, control topics should be managed by a dedicated broker (or sub-cluster of them) and vThing data topics must be partitioned on a set of different brokers. Accordingly, the system messages and components should be upgraded to support such a multi-broker configurations.

5 Security

In this section we address the security topic in the frame of our Fed4IoT architecture. To do so we evaluate the different interactions that can be performed among the components of the architecture from both an internal and external point of view so that we can extract security requirements to later introduce a series of technologies that address these issues.

5.1 Introduction and Motivation

Figure 34 presents a diagram of our VirIoT architecture which illustrates the interactions among its components and draws its security perimeter. We have classified the interactions in two different groups: internal (blue) and external (green) communications. Internal communications and external communications call for different requirements in terms of security.

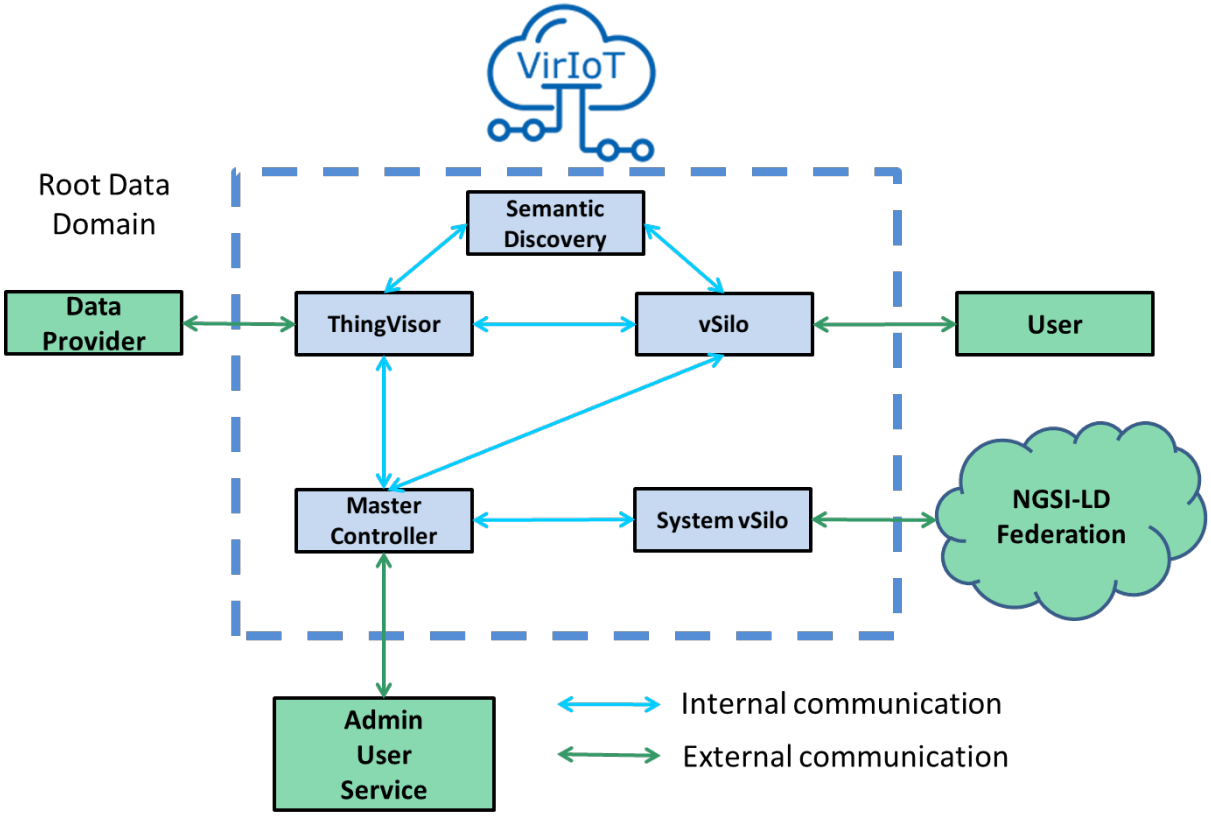

Figure 34: Main interactions of the VirIoT platform

While internal communications are performed inside a security domain conceptually owned by a single stakeholder, and therefore, between components whose functionality has been defined and developed inside a controlled system, with a set of well-known

APIs and endpoints that can be used, external communications occur between the inner VirIoT components and other external components such as Data Producers, VirIoT Users/Tenants or external platforms whose behaviour cannot be controlled, and whose communication payloads may traverse the Internet and be eavesdropped or manipulated by malicious users or services. For this reason, the requirements raised by internal communications are different from those raised by external ones.

Securing internal communications may just require, on the one hand, integrity of internal messages, which can be achieved by the use of certificates and SSL/HTTPS. Focusing, on the other hand, on the interactions between users/admin or services and the Master Controller, we must manage the access control to the latter, so that only authenticated and authorised entities can perform specific sets of actions, according to access control policies (e.g. by using JWT-based Access Control or Role-Based Access Control).

Additionally, since VirIoT is a virtualization platform, our attack surface is extended by the fact that vSilos are under the control of Tenants, which are non-trusted entities from VirIoT's perspective, and may maliciously try to inject fake messages either in the control or in the data internal planes. For this reason we introduce data-centric digital signatures of the messages, to let us refuse to acknowledge and discard fake information.

Much differently, external communications, such as the communication between Consumers and vSilos or between System vSilo and a federation of platforms corresponding to different domains, are exposed to more dangerous threats which require to strengthen the security means, by providing not only an access control mechanism, but also data integrity and privacy in a less controlled environment.

Finally, the communication between Providers/Producers and ThingVisors deserves special mention because it lets us define mechanisms to audit what Providers are integrated into our VirIoT platform, with the goal to also define an agreement with them. This triggers a different set of requirements, which can be dealt with by means od Distributed Ledger Technologies (DLT) and the use of Smart Contracts, for instance.

In this deliverable we first analyse the different technologies which can fit the security requirements of both internal and external communications. The following sections provide different technologies which cover the security aspects and topics introduced above, such as authentication, authorisation, privacy, integrity and DLT, among others.

5.2 Authentication, Identity Management and Access Control

According to the diagram presented before, users and services should be able to access to the VirIoT platform, and more specifically to operate with the Master Controller. In order to control the actions performed by these users and services, a first step is required. We need a mechanism that let us register the different users this platform have so that only authenticated users must be the ones allowed to operate over the VirIoT platform. Additionally, as we are explaining in the following section, our VirIoT platform also requires mechanisms or technologies that allow us to manage the access to the Master Controller and System vSilo.

Identity Managers are entities which traditionally has functionalities such as user/identity profile management or Single Sign On (SSO). This basic functionality has been enriched in the latest years with other focused on the authentication process and the privacy preserving mechanism so that in a security framework it became an anonymous credential system that ensures user privacy and minimal disclosure of personal information when accessing IoT services.

There are well-known enablers such as KeyRock[10] or One-IdM[11] which implement this functionality representing identities with a set of attributes. The former offers also the capability of being integrated with and access control management system which is explained in the following section 5.2.2 so that we can provide fine grained access to the resources of our system.

Because of the heterogeneous nature of IoT devices and networks, most of the recent access control proposals have been designed through centralized approaches in which a central entity or gateway is responsible for managing the corresponding authorization mechanisms, allowing or denying requests from external entities. This is actually the fist solution we used in VirIoT and that is discussed in section 5.2.1.

On the one hand, the inclusion of a central entity clearly compromises end-to-end security properties, which is considered an essential requirement [20] on IoT, because of the sensitivity level of potential applications. Besides, the dynamic nature of IoT scenarios with a potential huge number of connected devices complicates the access controls management with the central entity, affecting scalability. Accordingly, in section 5.2.3 we describe a more distributed solution, which also makes it possible to define access control rules at a finer level (e.g. externally accessing the information stored in the System vSilo component from an external federation or third party). Since this technology uses for the definition of access control policies triplets following this representation <subject, resource and action>, it let us specify fine-grained access control such as <serviceX, System vSilo, entityX> <serviceX, System vSilo, attributeY of entityX, attributeX of entityX> which, unlike JWT-based Access Control provides a richer definition and more flexible access control management.

5.2.1 JWT based Access Control

The JSON WEB Token (JWT) based access control is the simplest access control solution we implemented (as defined in RFC7519). It has been already introduced in D3.1 when we described VirIoT procedures. These procedures will be updated in the next release of the D3.x series deliverable (D3.2) to include also other access control schemes here discussed, such as XACML or DCapBAC, that currently are in a preliminary design phase.

JWT is a fairly recent standard (year 2015), which allows a backend system to encode certified *claims* in a payload to be transmitted as a JSON object. A claim is a `name:value`

[10] KeyRock: https://fiware-idm.readthedocs.io/en/latest/

[11] One-IdM: https://forge.fiware.org/plugins/mediawiki/wiki/fiware/index.php/Identity_Management_-_One-IDM_-_User_and_Programmers_Guide

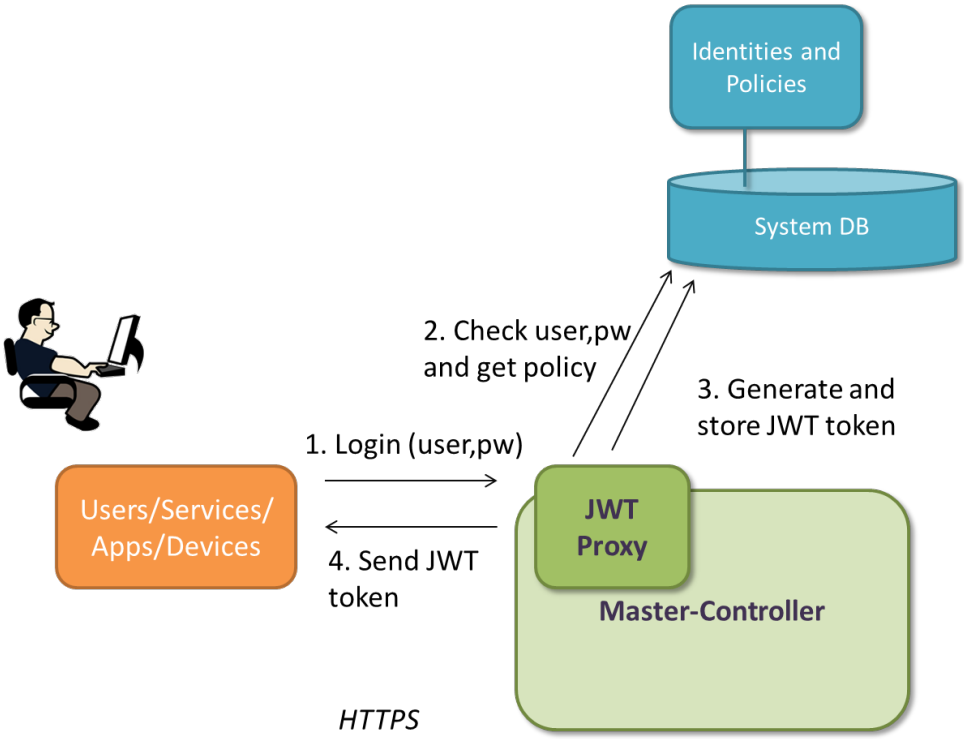

Figure 35: JWT based login

tupla and all claims form a JSON object, called JWT token, which is digitally signed and sent back to the client by the backend system during an authorization phase, e.g. a login procedure. The digital signature can use a secret (with the HMAC algorithm e.g. HS256) or a public/private key pair using RSA or ECDSA. As shown in the following JSON object, the JWT payload contains i) mandatory/public claims defined by IANA, such as Issued At (iat), Expiration Time (exp), JWT ID (jti), etc.; and ii) private claims to be used within a specific backend system such as 'identity' and 'user_claims' for our VirIoT platform.

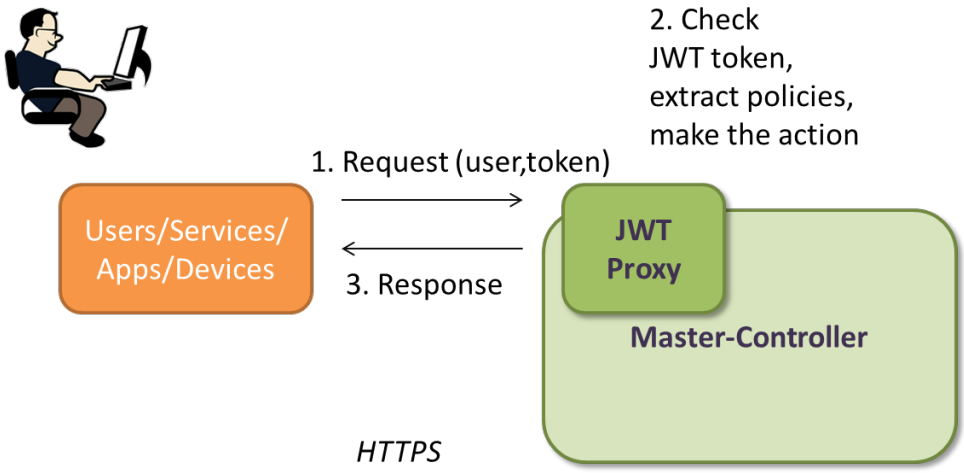

Figure 36: JWT based Access Control

JWT token payload

```
{
        'iat': 1568105212,
        'nbf': 1568105212,
        'jti': '060e7128-7cde-40da-9731-e1c3439bc926',
        'exp': 1572425212,
        'identity': 'admin',
        'fresh': False,
        'type': 'access',
        'user_claims': '{"role": "admin"}'
}
```

The token is packaged as `token=b64urlEnc(header).b64urlEnc(payload).b64urlEnc(signature)` and the client will use it in the HTTP Authorization Header from now on for subsequent calls to the backend system, thus informing it of who is making the call. This will allow the system to already have the authentication information directly in the token itself, thus avoiding having to go through a database or use the sessions to store the authentication information.

HTTP Header

```
Authorization: Bearer eyJ0eXAiOiJKV1Qi...<snip>...hHcIHOU"
Accept: application/json
Content-Type: application/json
```

Figure 35 and Figure 36 show how JWT access control scheme is used in the VirIoT system for the interaction with the master controller. The Master Controller uses a JWT proxy that intercepts HTTPs requests. The user (tenant) credentials are stored in an identity/policy store, which is merely implemented as a MongoDB collection within the SystemDB that contains the tuples <UserID,Hash(pwd),role,JWT token>.

In the login phase, the user pass username and password and the JWT proxy validate them interacting with the system database that contains user identities and associated policy. Currently, JWT based access control allows only user-related policy, and a user can have the role of 'user' or 'admin'. The access control policies related to these two figures allow them to perform the following control operations:

- Administrator: add vSilo Flavor, add ThingVisor, create his vSilo, access his vSilo, add/del vThing from his vSilo, listing and inspecting operations for all vSilos
- User: create his vSilo, access his vSilo, add/dev vThing from his vSilo, listing and inspecting operations for his vSilos

Besides validation, the JWT proxy generates and sends back to the user the JWT token embedding the 'identity' and 'user_credential' claims. From now on the user will use this token in next iterations with the Master Controller, which request VirIoT operations such as add vThing, create vSilo, etc.

5.2.2 Policy-based Access Control - XACML

The eXtensible Access Control Markup Language (XACML) [21] is a standard, declarative and XML-based language to express access control policies, which allows specifying the set of subjects which can perform certain actions on a specific set of resources, based on their attributes. Under the XACML data model, the definition of access control policies is mainly based on three elements: PolicySet, Policy and Rule. A PolicySet may contain other PolicySets and Policies, whereas a Policy includes a set of Rules, specifying an Effect (Permit or Deny), as a result of applying that Rule for a particular request. The Target sections of these elements define the set of attributes from resources, subjects, actions and environment to which the PolicySet, Policy or Rule are applicable. Moreover, since different Rules might be applicable under a specific request, XACML defines Combining Algorithms in order to reconcile multiple decisions. In addition, a set of obligations (Obligations class) can be used to notify a set of actions to be performed related to an authorization decision. Figure 37 below presents the XACML Policy Language Model as explained in the previous paragraphs.

XACML architecture consists mainly of four elements:

- Policy Administration Point (PAP): it is used to create a policy or set of policies
- Policy Decision Point (PDP): it evaluates applicable policies and makes authorization decisions

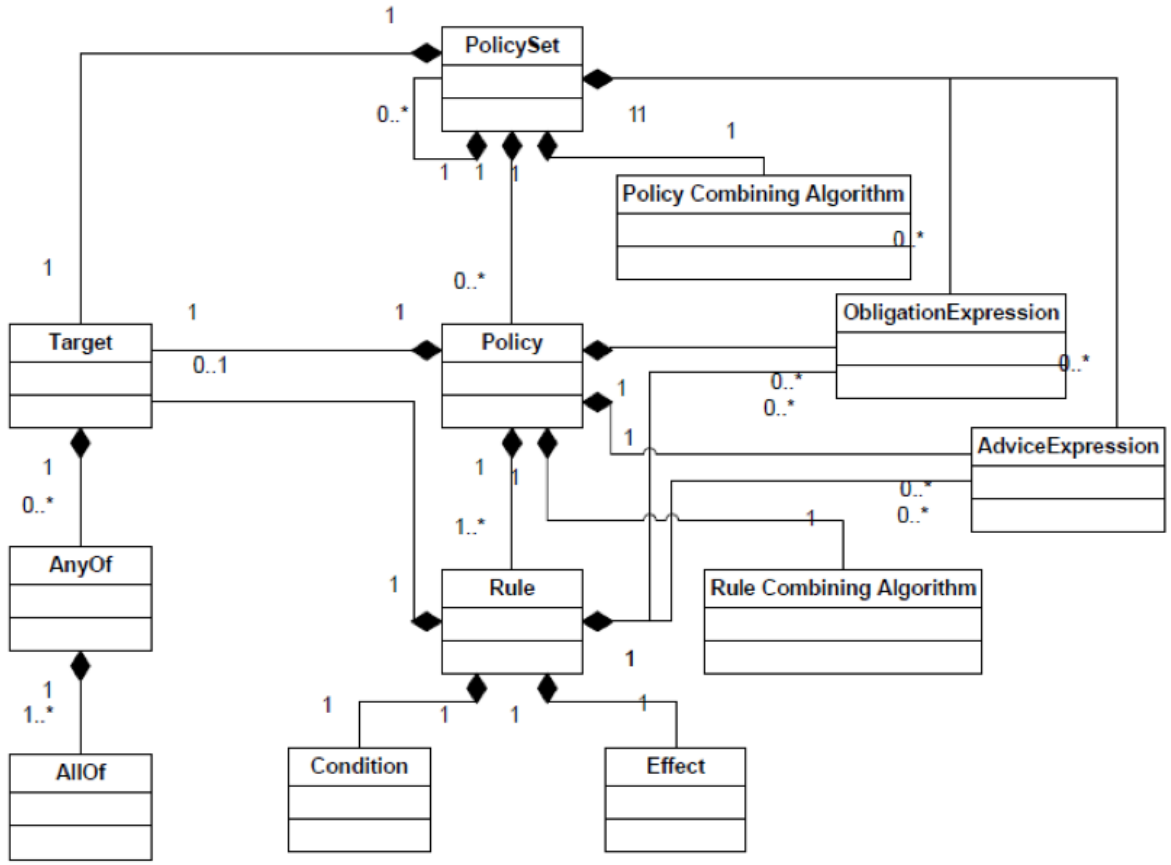

Figure 37: XACML Policy Language Model

- Policy Information Point (PIP): it acts as a source of attribute values
- Policy Enforcement Point (PEP): it is responsible for performing access control, by making decision requests and enforcing authorization decisions

Finally, the main interactions between these components under XACML standard are shown in Figure 38.

In light of this diagram, we can see that the different security policies which must prevail over a certain system are defined by the PAP (1. Define policy). When a certain service request accesses to a specific resource, it must issue a request message (2. Request resource). This request arrives to the PEP which is the one which executes the security policies and grants/denies the access to the resource to the service depending on them. To do so, it forwards the service request (3. Forward request) to the PDP. Nevertheless, the PDP could need extra information in order to make the security decision; this is the reason why a Context Handler is the central element of this diagram. It will receive each request (3 – 11) and forward it to the specific component the purpose of which is to provide all needed information to the PDP to generate the security response associated to

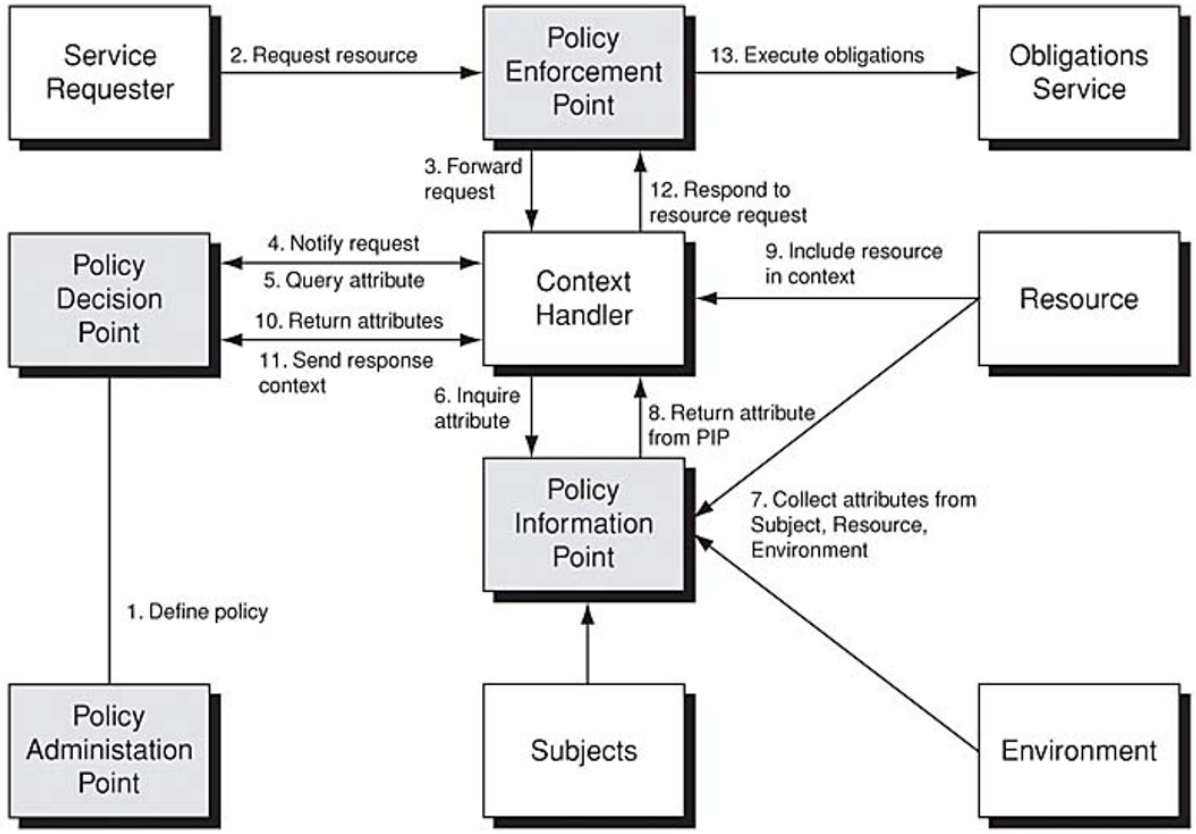

Figure 38: XACML standard overview

the message 3. Finally, such response is received by the PEP which enforces the security policy to the initial resource request.

Policy Administration Point

As described below, the PAP is responsible for generating the different security policies. The Figure 39 below presents a Web interface which is used by the security administrator to define the XACML security policies.

It counts with two administration sections (left panel): the first one to define security policies and the second one to define the different attributes that are managed by the former ones. In addition, there are some configuration aspects (right panel) that can be modified by the last button. Figure 40 shows the relation between Subject, Resource and Action attributes generation, defining them as generic pieces that will shape the policies. These generic attributes are created implementing any predefined or even customized XACML AttributeIDs that will enrich the expressiveness of the policies.

In this sense the possibility of defining customized policies has been increased and, furthermore, the use of XACML obligations has been also considered and implemented as part of the policy definition that will be used when generating Capability Tokens. This

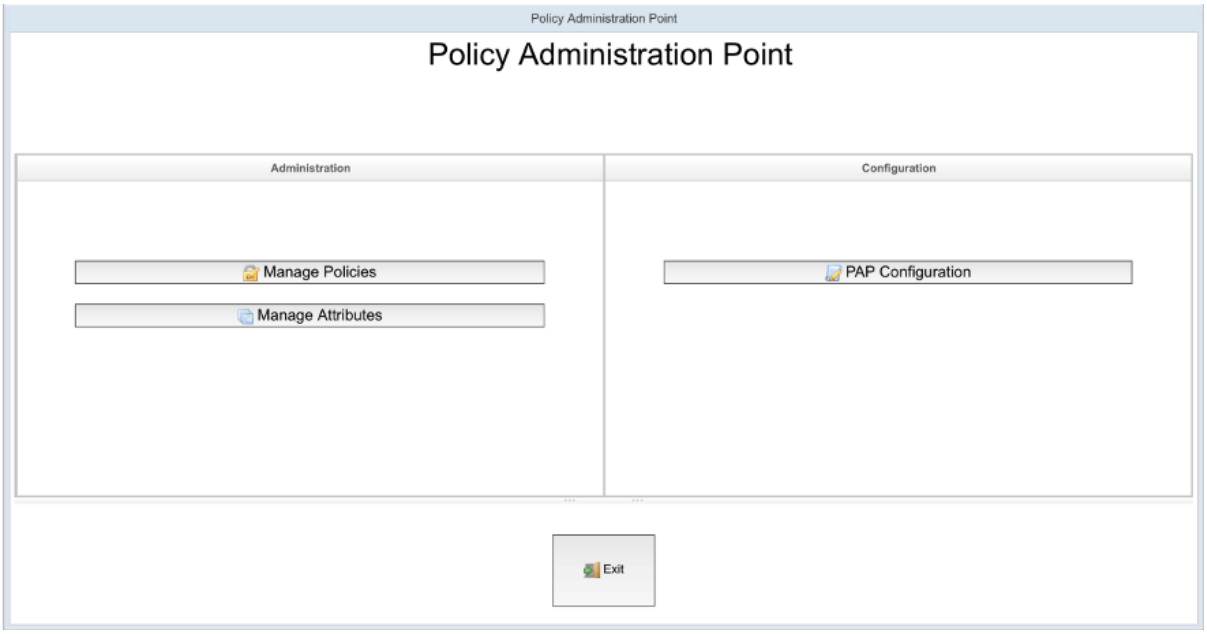

Figure 39: PAP Main View

way, the Capability Manager will be informed about the lifetime of the token associated with the specific policy.

Once the attributes for a Resource are defined, the attributes for Subjects and Actions can be specified too (see Figure 41 above). A policy contains rules that express the actions that are allowed or denied. For instance, we can create a policy with one rule and specify the resource it is referring to, what action can be performed with that resource and the subject that is allowed to do that. With the specified policies, a request can be made to the PDP that has the location of the policies. The PDP evaluates the request to decide PERMIT or DENY, returning the corresponding Obligation, if any, when supported. The request is sent encoded in JSON [22], which provides a less verbose representation of the information than other formats like XML, and improves the request processing as well. If such request is successfully resolved, the PDP will return a response in the form of {*"Result":"Permit"*}. Regarding its implementation, the PDP is deployed as a web service to be accessed by the entity acting as a PEP through the exchange of HTTP messages with JSON payloads containing the XACML requests or responses.

Policy Decision Point

The PDP is based on web technologies to make a scalable and lightweight solution. This PDP can be applied to any large scale deployment that requires XACML as policy language. The PDP evaluates XACML policies in XML representation. However, the requests are sent in JSON format, which improves the comprehension and the performance in the exchanging of requests and responses. In [23] a comparison of the XACML PDP and other PDP solutions are performed. This comparison shows that XACML PDP

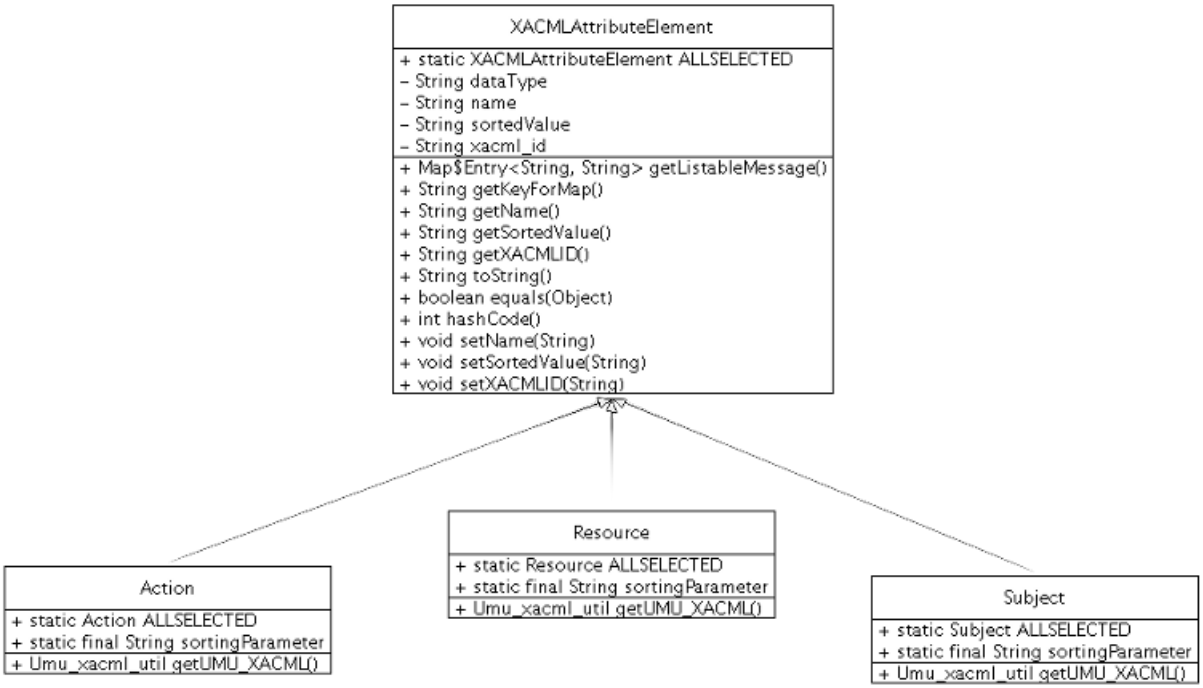

Figure 40: Generalization of the AttributeID

achieves the performance improvements over existing solutions in terms of scalability and efficiency.

5.2.3 Distributed Capability-Based Access Control

In a distributed approach, constrained devices (e.g. sensors, actuators) can make authorization decisions without the need to delegate this task to a different entity. In this case, end-devices are enabled with the ability to obtain, process and transmit information to other entities in a protected way. However, in a fully distributed approach, the feasibility of the application of traditional access control models, such as Role-Based Access Control (RBAC) or Attribute-Based Access-Control (ABAC), has not been demonstrated so far. Indeed, as previously mentioned, such models require a mutual understanding of the meaning of roles and attributes, as well as complex access control policies, which makes challenging the application of them on resource-constrained device. In that sense, it is necessary to consider the amount of computational resources which can be available on the end-device, since it may not be sufficient to apply a complex access control mechanism. Moreover, the impact of the potential applications of IoT in all aspects of our lives is shifting security aspects from an enterprise-centric vision to a more user-centric one. Therefore, usability becomes a key factor to be considered, since untrained users must be able to control how their devices and data are shared with other users and services.

Recently, the Capability-Based Access Control model (CapBAC) has been postulated as a realistic and promising approach to be deployed on IoT scenarios [24]. Inspired

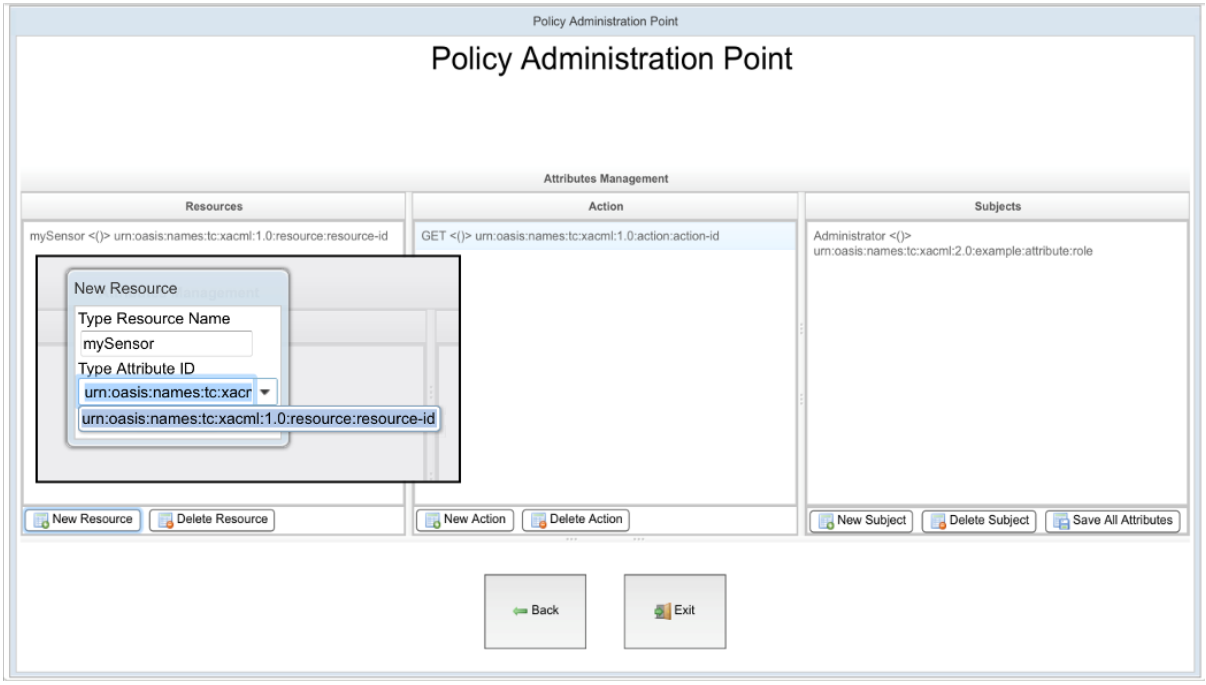

Figure 41: Defining different Attributes

by SPKI Certificate Theory [25] and authoriZation-Based Access Control (ZBAC) [26] foundations, CapBAC provides significant advantages over more established approaches by providing support for least privilege and preventing security problems such as the Confused Deputy problem [27]. Additionally, CapBAC simplifies user management tasks since the definition of complex policies is not required. The key element of CapBAC is the concept of capability, which was originally introduced by [28] as "token, ticket, or key that gives the possessor permission to access an entity or object in a computer system". This token is usually composed by a set of privileges which are granted to the entity holding the token.

In a typical CapBAC scenario, an entity (subject) tries to access a resource of another entity (target). Usually, a third party (issuer) generates a token for the subject specifying which privileges it has. Thus, when the subject attempts to access a resource hosted in the target, it attaches the token which was generated by the issuer. Then, the target evaluates the token, granting or denying access to the resource.

Therefore, a subject which wishes to get access to some information from a target, requires sending the token attached to the request. Thus, the target that receives such a token already knows the privileges that the subject has; only a local Policy Decision Point (PDP) further needs to verify the validity of the capability token. This procedure simplifies the access control mechanism, and happens to be a relevant feature in scenarios with resource-constrained devices since complex access control policies are not required to be deployed on constrained end-devices.

Additionally, the token must be tamper-proof and unequivocally identified in order to be considered in a real environment. Therefore, it is necessary to consider suitable cryptographic mechanisms to be used even on resource-constrained devices which enable an end-to-end secure access control mechanism. Moreover, given its high level of flexibility, CapBAC could be also extended to consider contextual conditions related to parameters which are locally sensed by end-devices. Also, this model could be complemented with other access control models by providing automated tools to infer the privileges to be embedded into the token.

Distributed CapBAC operation

The basic operation of our proposed DCapBAC is shown in Figure 42. Below, we clarify the different steps of the access control process.

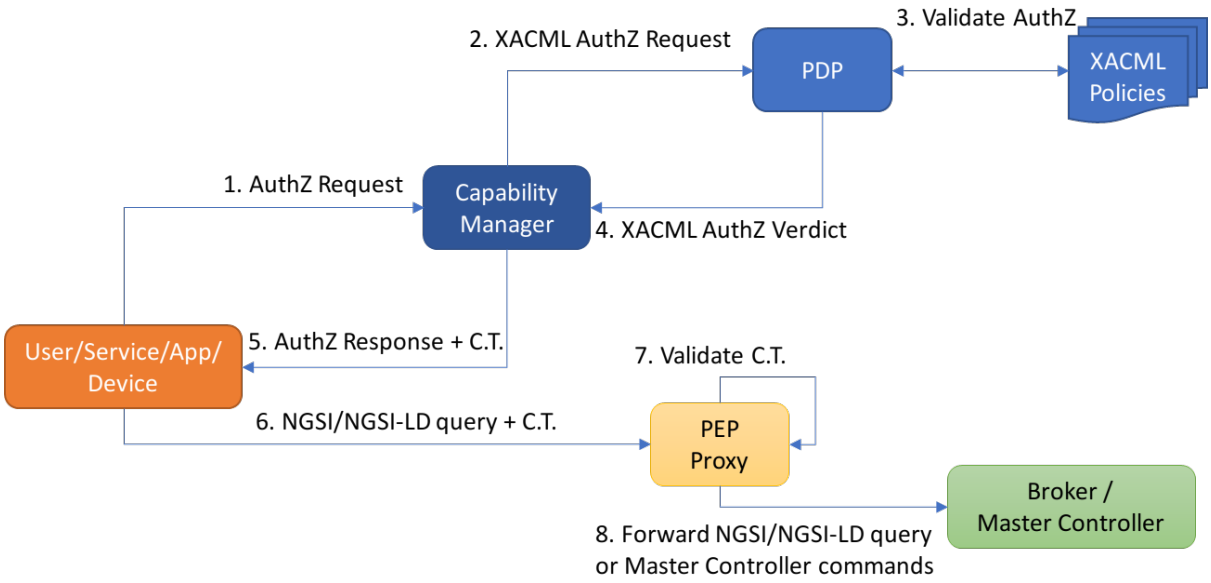

Figure 42: DCapBAC Operation Model

- Service Issue authorisation request. In this initial step, the service, application or device issue an authorisation request which is handled by the Capability Manager.
- Authorisation validation. The Capability Manager translate this authorisation request into a XACML authorisation request which is in turn validated by the PDP against the XACML policies file.
- XACML verdict. After a positive validation, the PDP issues a positive verdict, which is received by the Capability Manager.
- Issue Capability Token. The Capability Manager issues a capability token to the Subject to be able to access that resource.

- Access Request. Once the Subject has received the capability token, it attempts to access the device. For this purpose, it generates a request, in which the token is attached. This access request is handled by the PEP Proxy which validates the capability Token.

- Access to the broker. After a positive validation of the Capability Token, the PEP Proxy forwards the query to the broker, as well as it forwards its response to the Service, Application or device.

Capability token

In order to implement our distributed capability-based access control approach, we chose JSON as format to represent the capability token because of its suitability in constrained environments, such as those suggested by IoT scenarios. The next figure shows a capability token example, which has been used to demonstrate the feasibility of our proposal. Below, a brief description of each field is provided.

Example of Capability Token


```
{
        "id": "0h7be34m_0q2cx-7",
        "ii": 1369300359",
        "is": "jamartinez@odins.es",
        "su": "LNNh3/iyzZ/PQbBsuD+joYmTncm=G2O3xRD/3R7IzsJPiix9brrulTC=",
        "de": "http://platform.odins.es/",
        "si": "uPTor1jxykFQUGxXnRVRm01+uZM=kebPXS9VERYSyX4VFeHH9gW/yQT=",
        "ar": [
            {
                "ac": "GET",
                "re": "temperature",
                "f": 1,
                "co": [
                    {
                        "t": 5,
                        "v": 25,
                        "u": "Cel"
                    },
                    {
                        "t": 6,
                        "v": 21,
                        "u": "Cel"
                    }
                ]
            }
        ],
        "nb": 1369300359,
        "na": 1369300500
}
```


- Identifier (ID) (16 bytes). This field is used to un-equivocally identify a capability token. A random or pseudo-random technique will be employed by the issuer to ensure this identifier is unique.
- Issued-time (II) (10 bytes). Following the notation of [26], it identifies the time at which the token was issued as the number of seconds from 1970-01-01T0:0:0Z.
- Issuer (IS) (variable size). Entity issuing and signing the capability token.
- Subject (SU) (56 bytes). It makes reference to the subject to which the rights from the token are granted. A public key has been used to validate the legitimacy of the subject. Specifically, it is based on ECC-based public-key cryptography; therefore, each half of the field represents a public key coordinate of the subject using Base64.

- Device (DE) (variable size). It is a URI used to unequivocally identify the device to which the token applies.
- Signature (SI) (56 bytes). It carries the digital signature of the token. As a signature in ECDSA is represented by two values, each half of the field represents one of these values using Base64.
- Access Rights (AR). This field represents the set of rights that the issuer has granted to the subject.
- Action (AC) (variable size). Its purpose is to identify a specific granted action. Its value could be any CoAP method (GET, POST, PUT and DELETE).
- Resource (RE) (variable size). It represents the resource in the device for which the action is granted. – Condition flag (F) (1 byte). Following the notation in [28], it states how the set of conditions in the next field should be combined. A value of 0 means AND and a value of 1 means OR.
- Conditions (CO). Set of conditions which have to be fulfilled locally on the device to grant the corresponding action.
- Condition Type (T) (1 byte). The type of condition is verified as stated by [28].
- Condition value (V) (variable size). It represents the value of the condition.
- Condition Unit (U) (variable size). It indicates the unit of measure that the value represents.
- Not Before (NB) (10 bytes). NB expresses a time value. Before NB the token must not be accepted. Its value cannot be earlier than the II field and it implies the current time must be after or equal than NB.
- Not After (NA) (10 bytes). NA represents the time after which the token must not be accepted.

5.2.4 Shi3ld framework: An access control framework for RDF stores

In this section we describe an alternative technology that we have explored to implement access control, which is more grounded in RDF and ontologies, thus using existing software machinery and data models for semantically evaluating queries, in order to resolve access control requests.

The Shi3ld [29] project adopts Semantic Web languages at its core, and it reuses existing proposals, without adding new policy definition languages, parsers or validation procedures. It provides protection up to the level of triples. However, the work does not provide yet another context ontology: the model includes base classes and properties only, and delegates refinements and extensions to domain specialists. The framework assumes the trustworthiness of the information sent by the user, including data describing context

(e.g. location, device features, etc). The Shi3ld model is grounded on two ontologies, as described in figure 43: S4AC deals with core access control concepts and PRISSMA focuses on the user context. The access control model is built over the notion of Named Graph, thus supporting fine-grained access control policies, including to the triple's level. Enforcing permission models is an envisioned use case for RDF named graphs.

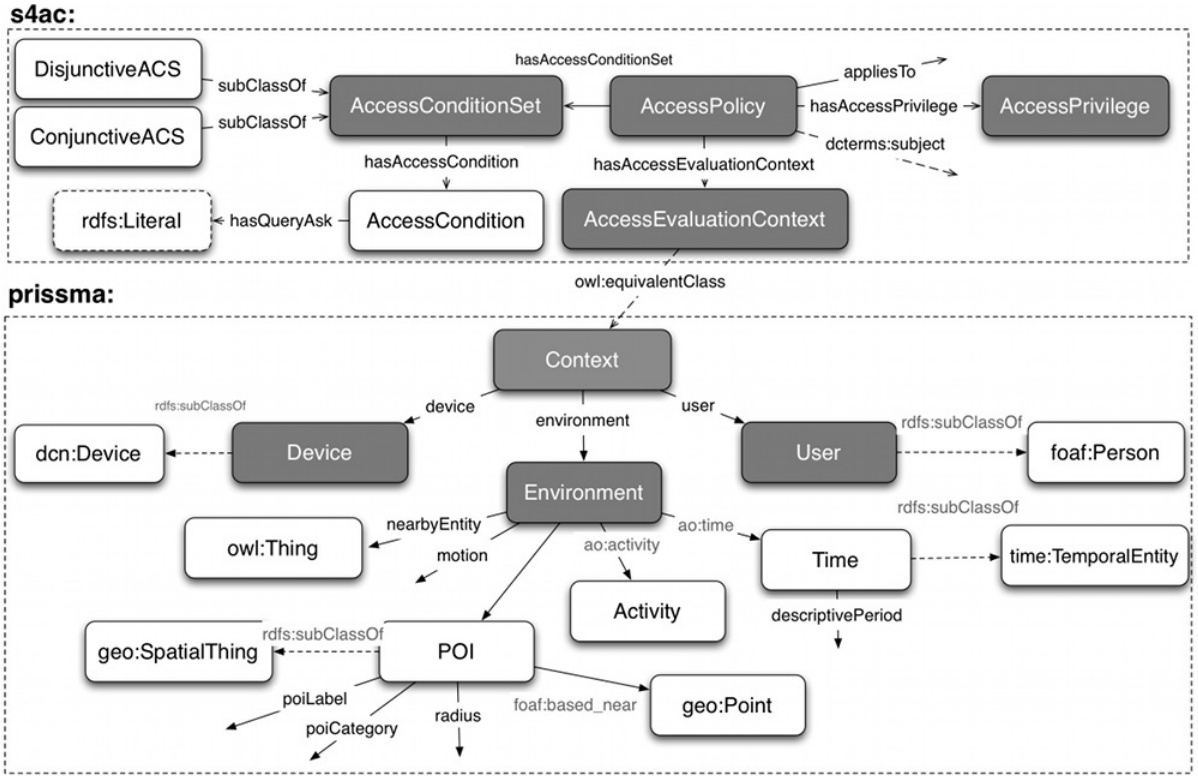

Figure 43: The Shi3ld model at a glance (grey boxes represent core classes).

The shield relies on named graphs to avoid depending on documents (one document can serialize several named graphs, one named graph can be split over several documents). As stated by the authors, their policies can be considered as access control conditions over g-boxes (according to W3C RDF graph terminology), with semantics mirrored in the SPARQL language. The S4AC vocabulary reuses concepts from SIOC, SKOS, WAC, SPIN.

Shi3ld is designed as a pluggable component for SPARQL endpoints. The access control flow is described below:

- The user queries the SPARQL endpoint to access the content. Context data is sent with the query and cached as a named graph using SPARQL 1.1 update language statements. Each time a context element is added they use an INSERT DATA, while they rely on a DELETE/INSERT when the contextual information is already stored and has to be updated. Summarizing, the client sends two SPARQL queries: the first is the client query to the data store, the second provides contextual information.

- The client query is filtered by the Access Control Manager instead of being directly executed on the SPARQL endpoint.
- The Access Control Manager selects the set of policies affecting the client query, i.e. those with a matching Access Privilege. This is achieved by mapping the client query to one of the four Access Privileges defined by S4AC with the SPIN vocabulary. The Access Conditions (SPARQL ASK queries) included in the selected policies are executed. According to the type of Access Condition Set (i.e., conjunctive or disjunctive), for each verified policy, the associated named graph is added to the set of accessible named graphs.

Figure 44 is used by the authors to illustrate the control flow described above:

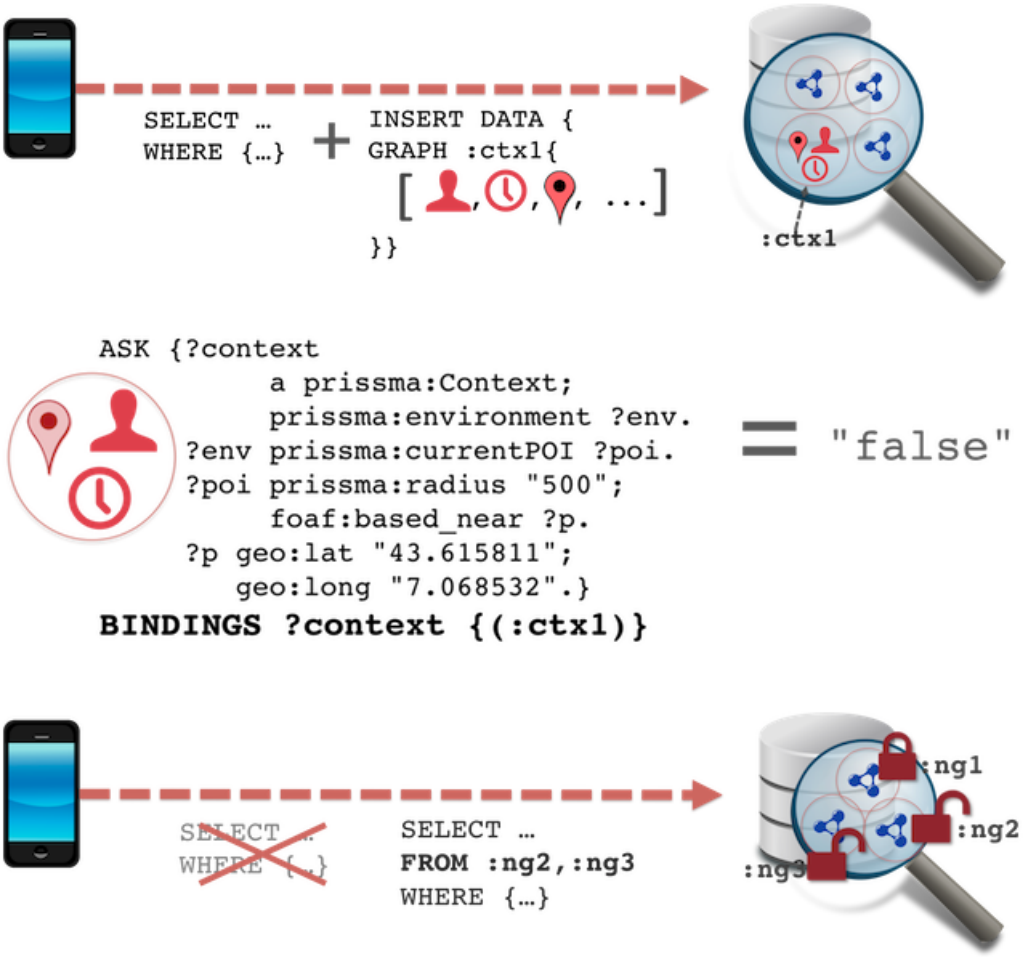

Figure 44: The scenario of access control enforcement in the Shi3ld architecture

5.3 Data-centric Security

As discussed above, protection of data that relies on perimeter security (access control to data servers) plus secure internal communication channels is appropriate for tightly closed deployments, where each component belongs to the same trusted zone. But users and stakeholders (especially in IoT ecosystems) are, increasingly, **not contained within the perimeter** of corporate networks anymore, and/or they cannot fully be trusted.

In our specific case, VirIoT's attack surface includes the vSilos, whose Broker are under direct control of Tenants, thus at risk of breaches from the inside, e.g. aimed to inject malicious information in the internal MQTT data or control plane, to create fake vThing data, to destroy vSilos of other tenants, etc. Consequently, we use data-centric security techniques, based on digital signatures of the payloads, in order to protect them both internally and externally, i.e. both from compromised, maliciously operated vSilos and from non-certified data producers of the Root Data Domain, as depicted in Figure 45.

Generally speaking, we have to ask how much closed can the VirIoT IT infrastructure be perceived by external Producers, Consumers, Tenants and externally federated Platforms, given that data might be:

- stored at multiple places
- shared while in-transit
- being worked by many components, prior to consumption

For instance, here follows a list of typical scenarios which show the broad range of situations where the VirIoT infrastructure would be perceived as an open deployment, endangdering its ability to protect data.

- Breaches from the inside are **likely** to occur, because vSilos are virtual-machines or containers running inside our perimeter, but under the authority of Tenants, who may act maliciously, trying to impersonate the Master Controller or a ThingVisor, injecting fake control or data messages.
- VirIoT **copies** sensor data to Big Data centers and analytics clusters, or Fed4IoT subcontractors, that manage data remotely on their personal devices and cannot be fully trusted, are authorized to get data. Data could be tampered with and data points be altered.
- There are GDPR/**regulations** that Fed4IoT, as a data controller and/or processor, must comply with, that require maintaining control of data no matter where it travels.
- VirIoT may want to implement its infrastructure on the **public cloud**, to save money.
- VirIoT may want to share private sensitive data coming from sensors with **different audiences**. Compliance with regulations is required, and each audience can have its own relevant data subset and restrictions on access.

5.3.1 Data-centric Integrity and JSON Digital Signatures

Thus, applications consuming data coming from VirIoT shall assume that data can possibly undergo forging or errors due to threats manifesting themselves within the boundaries of our platform, or where ever while in-transit, stored or processed, i.e. both internally and externally to VirIoT.

In all scenarios similar to the above ones, security must be able to protect data integrity wherever it is being used, viewed or saved. Towards this end, data-centric integrity approaches based on digitally signing relevant data pieces tend to be the preferred approach.

When we say that a node or component that is part of the VirIoT deployment is untrusted, or better said, it operates within the boundaries of a possibly untrusted zone, we are not proposing that the node is operated by malicious actors or dishonest service providers. But we acknowledge that it is part of the attack surface, hence we equip it with technologies that can guarantee, **by end-to-end contracts and verification, correctness of the data** it manages. In other words, data-centric integrity adds a security perimeter around data: security controls are embedded into data itself and they travel with the data at-rest, in-transit and at-work, towards the goal of enabling detection of data tampering and manipulation. End-users must be able to detect tampering, and VirIoT must seamlessly integrate with technology enacted towards this goal.

A typical scenario where meeting such requirement may be challenging is illustrated in Figure 45.

As a concrete example, we may assume that Producer 1 has traffic information in terms of the number of cars per minute entering city streets, while Producer 2 has information about air pollution of city streets. For instance, let the following data be produced by Producer 1, telling us that, at a particular data point, 22 cars per minute have entered Principal Avenue:

JSON traffic information in terms of cars per minute in a street.

```
{
 "id":"Principal Avenue",
 "type":"Street",
 "ObservedCars": {
  "type":"Property",
  "value":"22"
 }
}
```

Similarly, Producer 2 can tell us that (for instance at around the same time, we are exemplifying the scenario here) there are 50 micrograms per cubic meter of Particulate Matter PM25 in the same street.

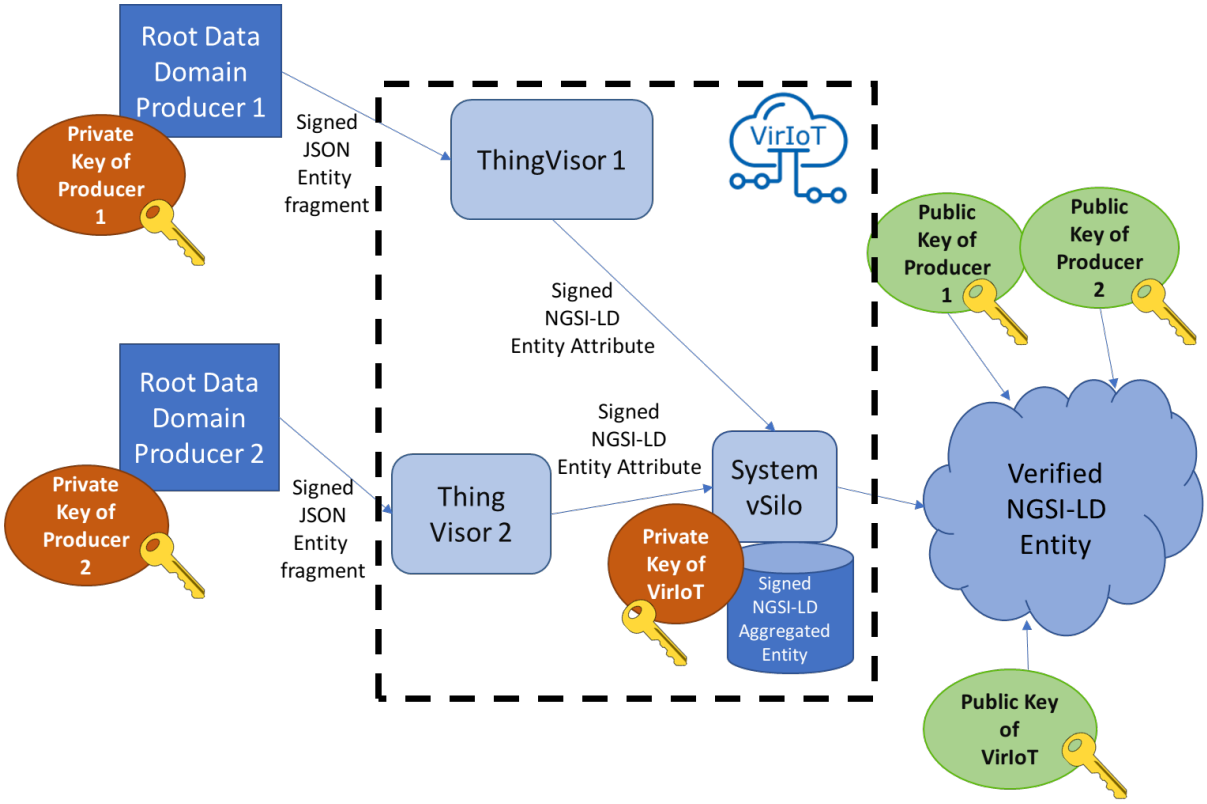

Figure 45: The scenario of two digitally-signed Entity fragments being aggregated by VirIoT

JSON pollution information in terms of PM concentration.

```
{
 "id":"Principal Avenue",
 "type":"Street",
 "ParticulateMatter": {
  "type":"Property",
  "value":"50"
 }
}
```

Developers of a novel smart city application would like to train a deep learning model on data, correlating traffic with PM25 concentration, but need the data points to be certified by the sensors producing them, in order to have a guarantee that **no bias or malicious stakeholders have influenced the resulting model**.

We are thus investigating techniques that allow to digitally sign JSON documents in plain text and maintain the validity of the signature even when the fields of the JSON payload are re-arranged by the intermediate brokers, or even aggregated with other JSON

documents.

One promising approach is the Linked Data Signatures 1.0 specification published by the W3C Digital Verification Community Group. This is an experimental specification and is undergoing regular revisions. Its goal is to add "authentication and integrity protection to linked data documents through the use of public/private key cryptography without sacrificing Linked Data features such as extensibility and composability" [30].

The key idea is to sign a a *canonicalized document* deriving from the original JSON, that is a deterministic, unique representation of the original JSON, since the original could have more that one possible representation. This process is sometimes also called *normalization*.

The first step is to interpret the original JSON as a Linked Data document, by creating a stable @context for it, so that it can be grounded into a RDF interpretation in terms of triples, because each key in the original JSON now has a precise meaning and is associated with a unique URI. We obtain the following JSON-LD document:

JSON-LD pollution information with example @context.

```
{
 "@context": {
  "id":"http://example.org/id",
  "type":"http://example.org/type",
  "value":"http://example.org/value",
  "ParticulateMatter":"http://example.org/PM"
 },
 "id":"Principal Avenue",
 "type":"Street",
 "ParticulateMatter": {
  "type":"Property",
  "value":"50"
 }
}
```

The second step is to apply the canonicalization algorithm, which takes as input the above document, that has more than one possible representation, and always transforms it into a deterministic representation, giving the following output:

RDF triples representing pollution information in normalized form.

```
_:c14n0 <http://example.org/type> "Property" .
_:c14n0 <http://example.org/value> "50" .
_:c14n1 <http://example.org/PM> _:c14n0 .
_:c14n1 <http://example.org/id> "Principal Avenue" .
_:c14n1 <http://example.org/type> "Street" .
```

We can see that a set of RDF statements is generated by the algorithm, which has created **two blank nodes** *_:c14n0* and *_:c14n1*, and has connected all statements to them. Statements are also alphabetically sorted so that the normalized representation is always the same, no matter the order of the fields of the original JSON, as long as the original keys and values stay the same.

The third step is to digitally sign the canonicalized document, as follows:

Digitally signed JSON-LD pollution information.


```
{
  "@context": [
    {
      "@version": 1.1
    },
    {
      "id": "http://example.org/id",
      "type": "http://example.org/type",
      "value": "http://example.org/value",
      "ParticulateMatter": "http://example.org/PM"
    },
    "https://w3id.org/security/v2"
  ],
  "id": "Principal Avenue",
  "type": "Street",
  "ParticulateMatter": {
    "type": "Property",
    "value": "50"
  },
  "proof": {
    "type": "RsaSignature2018",
    "created": "2019-12-24T09:31:03Z",
    "creator": "https://example.com/jdoe/keys/1",
    "jws": "eyJhbGciOiJQUzI1NiIsImI2NCI6ZmFsc2UsImNyaXQiOlsiYjY0Il19..
       AX4FFvQePKFKLPfKfoSpkgssC8gUL5Rahi-
       jaPoiDSB2T1cB76vwbVvArn8DeFMmfDy3TXCdYvGv0G8JjYSytUH-
       tWMPRkCHE3E53J6MBeJ_4YhULsNlcw1OOh8-6r-
       L6pwuOJUjkhWb43oZ15Da33KnZClV78bXwixTDjkcuHw",
    "nonce": "7e8032bb"
  }
}
```


The resulting signature is a type of proof, and is comprised of information about the signature, parameters required to verify it, and the signature value itself. A linked data signature typically includes at least the following attributes:

- type (required): A URI that identifies the digital signature suite that was used to

create the signature.

- created (required): The string value of an ISO8601 combined date and time string generated by the Signature Algorithm.
- nonce (optional, but strongly recommended): A string value that is included in the digital signature and MUST only be used once for a particular domain and window of time. This value is used to mitigate replay attacks.
- signature value (required): One of any number of valid representations of signature value generated by the Signature Algorithm.

5.3.1.1 Challenges in Fed4IoT

By the very nature of Linked Data, NGSI-LD automatically allows our platfomr to aggregate data points that refer to the same Entity. In the example above, both the traffic information and the pollution information refer to the same street. Thus, when the two distinct fragments are exported to the System vSilo, they get aggregated under the same Entity. We introduce a second layer of signature (see again Figure 45) so that:

- the sensors that originally produced the data points securely associate each value to the identifier of the street "Principal Avenue";
- VirIoT vouches for the aggregation process, digitally signing the resulting Entity, composed by two distinct attributes ("ParticulateMatter" and "ObservedCars").

By consulting the public keys of both the Producer sensors and of VirIoT, the Consumer application can be sure that the values have not been altered and that the platform has aggregated the value within a single Entity in a meaningful way, and that no one has injected other values into the same Entity afterwards.

We gain the following advantages:

- Non-repudiation
- Integrity
- Entities can be hosted on insecure servers
- Entities can be replicated and cached in several places

A second challenge is connected to the fact that, following this approach, we obtain a JSON-LD document which is a Regular NGSI-LD Entity + Signature.

As we have said, Signature follows the Linked Data Signatures specification. But if signature follows that W3C specification, the resulting document is not a valid NGSI-LD document (it is, of course, a valid JSON-LD document).

This is due to the fact that the Signature portion cannot be one-to-one mapped to a standard NGSI-LD Property. It uses a @context based on the following resources and vocabulary:

- https://w3id.org/security/v1 and v2
- https://web-payments.org/vocabs/security

Thus, unfortunately the current model for NGSI-LD Properties dictates that each Property must have a "type":"Property", while the "proof" field of the Signature has, for example, a "type":"RSASignature2018" (or other kinds, dictated by the above security vocabulary resources and @context).

So a simple addition of new fileds, such as the missing "creator", "nonce", "jws", to the core fields of a standard NGSI-LD CIM Property is not enough, as the W3C Signature cannot be interpreted as a ETSI CIM Property on a semantic level, so far.

Thus we are investigating the various possibilities to align the two standards and/or allow VirIoT to manipulate digitally signed JSON-LD documents. The main idea is to put the two standards side-by-side and create a straightforward bridging between them, which would be able to protect from current/future changes to the structure of the Linked Data Signature, that is currently evolving just like NGSI-LD is.

5.3.2 Data-centric Privacy and CP-ABE Access Control for Sensitive Data

External communication, as we already motivated at the beginning of this sections have more strict requirements in terms of security. These communications can contain sensitive information which not only must not be manipulated, but neither be accessible for external parties. For this reason, in the following section we provide a technology to broadcast information over a shared media in a privacy preserving way.

The requirements presented by common IoT scenarios require more flexible data sharing models between entities while the privacy of smart objects involved is still preserved. Unlike the current Internet, IoT interaction patterns are often based on short and volatile associations between entities without a previously established trust link. In this section, we review existing cryptographic schemes that can be potentially used to implement a secure information sharing based on the push model. These mechanisms should be applied on the smart objects themselves in order to provide an end-to-end secure data dissemination.

Attribute-Based Encryption (ABE) is gaining attention because of its high level of flexibility and expressiveness, compared to previous schemes. In ABE, a piece of information can be made accessible to a set of entities whose real, probably unknown identity, is based on a certain set of attributes. This represents a step forward in order to realize a privacy-preserving and secure data sharing scheme in pervasive and ubiquitous scenarios, since consumers do not need to reveal their true identity to obtain information, while producers can be sure that their data are accessed only by authorized entities.

Based on ABE, two alternative approaches were proposed. In KP-ABE [31], a cipher text is encrypted under a set or list of attributes, while private keys of participants are associated with combinations or policies of attributes. In this case, a data producer has limited control over which entities can decrypt the content, being forced to rely on the AA entity issues appropriate keys for getting access to disseminated information. In contrast,

in a CP-ABE scheme [32], a cipher-text is encrypted under a policy of attributes, while keys of participants are associated with sets of attributes. Thus, CP-ABE could be seen as a more intuitive way to apply the concepts of ABE; on the one hand, a producer can exert greater control over how the information is disseminated to other entities, On the other hand, a user's identity is intuitively reflected by a certain private key. In addition, CP-ABE is secure to collusion attacks, that is, different keys from their corresponding entities cannot be combined to create a more powerful decryption key. This feature is due to the use of individual random factors for each key generation. Moreover, in order to enable the application of CP-ABE on constrained environments, the scheme could be used in combination with Symmetric Key Cryptography (SKC) [33]. Thus, a message would be protected with a symmetric key, which would be encrypted with CP-ABE under a specific policy.

5.3.2.1 Policies for information sharing

CP-ABE policies indicate the set of entities that are enabled to decrypt the information to be shared by specifying the sets of attributes that these entities must satisfy, and they allow combining them using logical operators.

The resulting CP-ABE policy is used to encrypt the information to be shared by a dedicated CP-ABE engine subcomponent.

The following fragment shows an high-level example of a typical CP-ABE policy which, as we can see, is a logical AND combination of specific attributes that can be associated, for instance, to an entity: `policy = "role:admin AND company:OdinS"`.

5.4 Distributed Ledger Technology

In this chapter, we investigate a possibility to use Distributed Ledger Technology (DLT) to provide services according to agreements between users and service providers, that are captured in machine-readable and verifiable sandboxes.

This approach will allow us to track the information exchange between VirIoT and the Root Data Domain's actors that give data to VirIoT's ThingVisors. This exchange of information can be captured into machine-readable agreements, and automatic decisions, compensation and/or auditing can be performed, based on that.

DLT consists of at least two peers that store identical copies of a ledger each, and each peer individually updates the ledger. When a ledger is updated, identical copies of ledgers are correctly updated according to a consensus algorithm. A digital signature technology, encrypting the hash of the data by secret key, ensures the authentication and integrity of data. It makes possible to protect data from counterfeit. The concept of DLT is derived from Blockchain, however there is no clear unique definition on the difference between DLT and Blockchain. There are two types of Blockchain. Public Blockchain is one type of Blockchain which opens to any parties. Private Blockchain is another type of Blockchain which is accessible by the authorised limited parties. In this deliverable, when

we say DLT we mean Private Blockchain, because the access to DLT is to be limited to authorised parties, in our scenario.

Thus the use of DLT enables to protect from counterfeiting an "agreement" between two parties. Our idea is to use DLT to protect agreements for some specific service provisions of the VirIoT platform.

5.4.1 Smart Contract

Smart contract is a computer protocol to facilitate, verify, or enforce the negotiation or performance of a contract. Smart contracts allow the performance of credible transactions without third parties. As such, a DLT is suitable for smart contracts because of its decentralised architecture. Execution of contract and settlement are automatically processed by a computer program.

In order to support smart contracts by Blockchain/DLT, a smart contract is described by a specific transaction logic, which defines the conditions (input data and trigger) and settlements (actions), and is defined within chaincode. Multiple smart contracts with the same transaction logic can be defined within the same chaincode. All smart contracts are accessible via API for Blockchain, as described in Figure 46.

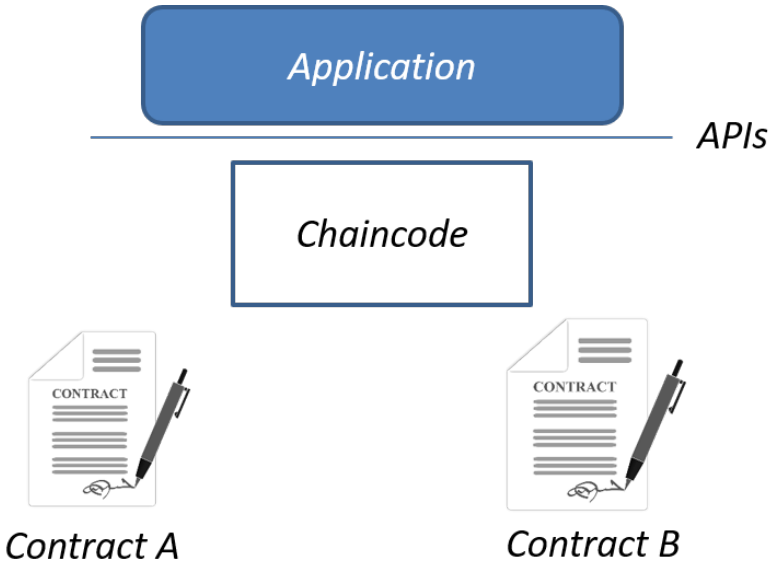

Figure 46: Smart contract with Blockchain

5.4.2 Smart contract for Fed4IoT Platform

Smart City applications using the Fed4IoT's VirIoT platform are required to access to the virtual IoT data which is collected within the data domains. In order to authorise the access to the Root Data Domain according to the contract between the Root Data Domain and Fed4IoT platform, it is required to define the transaction logic for such

authorisation process, defining the trigger of the process, acceptance criteria and the result of the process.

As an example, a smart contract enables that a root data domain actor can sell their virtual IoT data to the user via Fed4IoT platform. A smart contract function is supported by Fed4IoT platform to purchase their real IoT data. The prices for purchasing real IoT data par each type of data is provided by the root data domain actor and agreed between the root data domain and Fed4IoT platform. Based on this agreement, a transaction logic is defined by smart contract function.

When the user of Fed4IoT requests the virtual IoT data via vSilo, it triggers a smart contract transaction. The VirIoT platform then inputs user information and a set of types of data to smart contract function. Smart contract function then creates an contract based on the agreement between platform and root data domain. Possible counterfeiting of the contract is protected by Blockchain.

Once the contract is confirmed, the smart contract function authorises the user to access to the virtual IoT data collected by root data domain via the VirIoT platform and to initiate a money transaction to root data domain. A new contract or updated contract will be created when a new set of service is requested to Fed4IoT, using the same transaction logic as described in Figure 47.

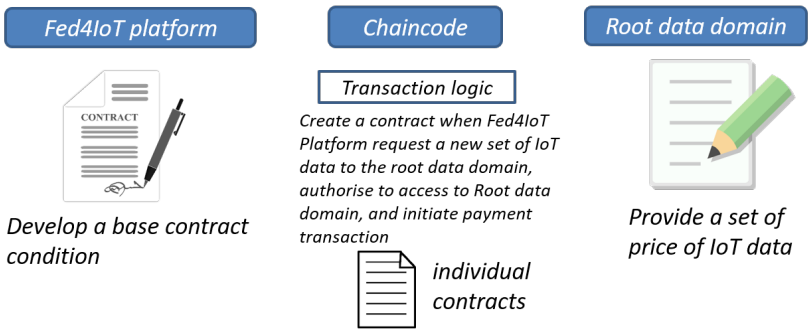

Figure 47: Exampled smart contract application to Fed4IoT

This approach, using smart contract functions, can be used for any kind of contract (e.g., service authorisation without payment), once a transaction logic is defined according to the business logic.

6 Conclusion

In this deliverable we have presented a first major revision of the Fed4IoT architecture that was originally introduced in deliverable D2.2.

The present revision concerns new components devoted to semantic registration of the data pieces flowing through the VirIoT platform, that we introduce in order to enable Tenants of the platform to carry out semantic searching for the relevant information that fits their Applications.

Our approach, based on a Semantic Discovery component and a System vSilo, additionally allows to connect VirIoT to external platforms, forming a large-scale federation.

We leverage the ETSI NGSI-LD standard, to which Fed4IoT is actively contributing, based on the findings of the project.

Experimental and simulation study of our topic-based pub/sub architecture for internal data distribution gave us useful insights about how we may restructure the topic names and the inner workings of control+data planes, for better efficiency.

Transversely, we have started to introduce security and privacy-dedicated technologies into our platform.

We expect to further develop our architecture, until we fully stabilize it in the next architecture deliverable (D2.3) iteration, while the next iteration of deliverable D4.x series will fully exploit the architecture and consolidate the design of the Information Sharing Layer and of the Security framework.

References

[1] ETSI GS CIM 009. Context Information Management (CIM): NGSI-LD API. [Online]. Available: https://docbox.etsi.org/ISG/CIM/Open

[2] W. Li, G. Tropea, A. Abid, A. Detti, and F. Le Gall, "Review of standard ontologies for the web of things," in *2019 Global IoT Summit (GIoTS)*. IEEE, 2019, pp. 1–6.

[3] IETF, "rfc8428 Sensor Measurement Lists (SenML)," https://tools.ietf.org/html/rfc8428, 2018.

[4] Un/edifact rec 20 – codes for units of measure used in international trade. [Online]. Available: http://www.unece.org/tradewelcome/un-centre-for-trade-facilitation-and-e-business-uncefact/outputs/cefactrecommendationsrec-index/code-list-recommendations.html

[5] J. Swetina, G. Lu, P. Jacobs, F. Ennesser, and J. Song, "Toward a standardized common m2m service layer platform: Introduction to onem2m," *IEEE Wireless Communications*, vol. 21, no. 3, pp. 20–26, 2014.

[6] E. Mosquitto. An open source mqtt broker. [Online]. Available: https://mosquitto.org/

[7] VerneMQ. Clustering mqtt for high availability and scalability. [Online]. Available: https://vernemq.com

[8] RabbitMQ. Rabbitmq. [Online]. Available: https://www.rabbitmq.com/

[9] HiveMQ. Reliable data movement for connected devices. [Online]. Available: https://www.hivemq.com/

[10] eMQTT. The massively scalable mqtt broker for iot and mobile applications. [Online]. Available: http://emqtt.io/

[11] Kubernetes (k8s): Production-grade container orchestration. [Online]. Available: https://kubernetes.io/

[12] H. Kwak, C. Lee, H. Park, and S. Moon, "What is twitter, a social network or a news media?" in *Proceedings of the 19th international conference on World wide web*. AcM, 2010, pp. 591–600.

[13] *MATLAB version 9.6.0.1114505 (R2019a) Update 2*, The Mathworks, Inc., Natick, Massachusetts, 2019.

[14] V. Setty, G. Kreitz, R. Vitenberg, M. Van Steen, G. Urdaneta, and S. Gimåker, "The hidden pub/sub of spotify:(industry article)," in *Proceedings of the 7th ACM international conference on Distributed event-based systems*. ACM, 2013, pp. 231–240.

[15] Wikipedia contributors, "Publish–subscribe pattern — Wikipedia, the free encyclopedia," 2019, [Online; accessed 17-October-2019]. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Publish%E2%80%93subscribe_pattern&oldid=917772811

[16] H. Liu, V. Ramasubramanian, and E. G. Sirer, "Client behavior and feed characteristics of rss, a publish-subscribe system for web micronews," in *Proceedings of the 5th ACM SIGCOMM conference on Internet Measurement*. USENIX Association, 2005, pp. 3–3.

[17] L. Breslau, P. Cao, L. Fan, G. Phillips, S. Shenker *et al.*, "Web caching and zipf-like distributions: Evidence and implications," in *Ieee Infocom*, vol. 1, no. 1. INSTITUTE OF ELECTRICAL ENGINEERS INC (IEEE), 1999, pp. 126–134.

[18] Y. Teranishi, R. Banno, and T. Akiyama, "Scalable and locality-aware distributed topic-based pub/sub messaging for iot," in *2015 IEEE Global Communications Conference (GLOBECOM)*. IEEE, 2015, pp. 1–7.

[19] Scalagent. (2015) Benchmark of mqtt servers. [Online]. Available: http://www.scalagent.com/IMG/pdf/Benchmark_MQTT_servers-v1-1.pdf

[20] T. Heer, O. Garcia-Morchon, R. Hummen, S. L. Keoh, S. S. Kumar, and K. Wehrle, "Security challenges in the ip-based internet of things," *Wireless Personal Communications*, vol. 61, no. 3, pp. 527–542, 2011.

[21] O. Standard, "extensible access control markup language (xacml) version 2.0," 2005.

[22] D. Crockford, "The application/json media type for javascript object notation (json), 2006," *URL http://tools. ietf. org/html/rfc4627*, 2006.

[23] L. Griffin, B. Butler, E. de Leastar, B. Jennings, and D. Botvich, "On the performance of access control policy evaluation," in *2012 IEEE International Symposium on Policies for Distributed Systems and Networks*. IEEE, 2012, pp. 25–32.

[24] S. Gusmeroli, S. Piccione, and D. Rotondi, "A capability-based security approach to manage access control in the internet of things," *Mathematical and Computer Modelling*, vol. 58, no. 5-6, pp. 1189–1205, 2013.

[25] J. L. Hernández-Ramos, A. J. Jara, L. Marin, and A. F. Skarmeta, "Distributed capability-based access control for the internet of things," *Journal of Internet Services and Information Security (JISIS)*, vol. 3, no. 3/4, pp. 1–16, 2013.

[26] J. Bradley, N. Sakimura, and M. B. Jones, "Json web token (jwt)," 2015.

[27] N. Hardy, "The confused deputy:(or why capabilities might have been invented)," *ACM SIGOPS Operating Systems Review*, vol. 22, no. 4, pp. 36–38, 1988.

[28] S. Li, J. Hoebeke, and A. Jara, "Conditional observe in coap," *Constrained Resources (CoRE) Working Group, Internet Engineering Task Force (IETF), work in progress, draft-li-core-conditional-observe-03*, 2012.

[29] L. Costabello, S. Villata, and F. Gandon, "Context-aware access control for rdf graph stores," vol. 242, 08 2012.

[30] W3C. Linked Data Signatures 1.0. Draft Community Group Report 18 October 2019. [Online]. Available: https://w3c-dvcg.github.io/ld-signatures

[31] V. Goyal, O. Pandey, A. Sahai, and B. Waters, "Attribute-based encryption for fine-grained access control of encrypted data," in *Proceedings of the 13th ACM conference on Computer and communications security*. Acm, 2006, pp. 89–98.

[32] J. Bethencourt, A. Sahai, and B. Waters, "Ciphertext-policy attribute-based encryption," in *2007 IEEE symposium on security and privacy (SP'07)*. IEEE, 2007, pp. 321–334.

[33] O. Garcia-Morchon, S. Kumar, S. Keoh, R. Hummen, and R. Struik, "Security considerations in the ip-based internet of things draft-garcia-core-security-06," *Internet Engineering Task Force*, 2013.